AFRL-RI-RS-TR-2008-20
Final Technical Report
January 2008

HOT DIFFUSION – TACTICAL INFORMATION MANAGEMENT SUBSTRATE

ATC-NY (Odyssey Research Associates)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2008-20 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
MICHAEL MUCCIO Work Unit Manager	JAMES W. CUSACK Chief, Information Systems Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JAN 2008	Final	Mar 06 – Sep 07

4. TITLE AND SUBTITLE

HOT DIFFUSION – TACTICAL INFORMATION MANAGEMENT SUBSTRATE

5a. CONTRACT NUMBER
FA8750-06-C-0066

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Matt Stillerman

5d. PROJECT NUMBER
ICED

5e. TASK NUMBER
06

5f. WORK UNIT NUMBER
03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ATC-NY (Odyssey Research Associates)
33 Thornwood Drive, Suite 500
Ithaca NY 14850-1280

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RISE
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-20

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB -08-0176

13. SUPPLEMENTARY NOTES

14. ABSTRACT
To summarize our results, ATC-NY developed a new suite of algorithms for information dispersion in HotDiffusion. Our testing indicates that, in comparison with more conventional approaches, HotDiffusion should excel under very sparse network conditions, especially if there is enough dynamism in connectivity. We expect its performance to degrade gracefully as conditions worsen. We measured performance using a full implementation of HotDiffusion, as well as simulation. The implementation runs on a testbed of handheld wireless nodes. It also runs on a conventional wired network with emulated wireless links.

15. SUBJECT TERMS
Peer-to-peer, network, wireless, tactical, publish/subscribe, data diffusion algorithm

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Michael Muccio
U	U	U	UL	56	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

Lists of Figures

Abstract

Conditions in ad hoc tactical networks can be very challenging, with poor link quality, highly dynamic topology, and frequent network partitions. ATC-NY has created a prototype of HotDiffusion, a novel system for providing information management services in this environment. It is a peer-to-peer system, in which network nodes serve information to one another. Its novelty lies in the method of dispersion of information: Information objects diffuse among the nodes and replicate, eventually achieving a steady-state distribution. The project aimed to determine the feasibility of this concept, to uncover the necessary engineering principles, and to build and experiment with a prototype. Perhaps the greatest challenge was finding a method of controlling the dispersion process purely locally – without global coordination.

To summarize our results: ATC-NY developed a new suite of algorithms for information dispersion in HotDiffusion. Our testing indicates that, in comparison with more conventional approaches, HotDiffusion should excel under very sparse network conditions, especially if there is enough dynamism in connectivity. We expect its performance to degrade gracefully as conditions worsen. We measured performance using a full implementation of HotDiffusion, as well as simulation. The implementation runs on a testbed of handheld wireless nodes. It also runs on a conventional wired network with emulated wireless links.

1. Summary

ATC-NY has conducted a research effort to validate and develop a novel peer-to-peer concept for providing information management services in *ad hoc* tactical networks. The project aimed to determine the feasibility of this concept, to uncover the necessary engineering principles, and to build and experiment with a prototype. The notional system underlying this research, called *HotDiffusion*, is intended to function under the very challenging network conditions prevalent in *ad hoc* tactical networks. Information published by clients diffuses outward among the nodes. This cloud of replicas achieves a steady state distribution without global coordination. Clients seeking information will query their local cache, and will request information from their near neighbors.

Information dispersion consists of three concurrent activities: random migration of information objects between nodes, replication of objects, and expiration of objects. A major focus of our research has been the design and validation of algorithms for locally controlling the dispersion process to achieve the desired steady state distribution of information.

We built a prototype of HotDiffusion which runs on a testbed of mobile wireless nodes. The prototype also can run with emulated wireless communications.

To summarize, HotDiffusion appears to excel under very sparse network conditions, especially if there is enough dynamism in connectivity. We expect its performance to degrade gracefully as conditions worsen. It also functions, though not optimally, under good network conditions. HotDiffusion will opportunistically "ferry" information on nodes moving between disconnected sub-networks. It does this without planning or central coordination.

On theoretical grounds, we expect HotDiffusion to excel in circumstances where node motion and connectivity are less predictable, and where the quality of the network varies greatly over time, and geographically.

2. Introduction

ATC-NY has developed and validated a novel peer-to-peer concept for providing information management services in *ad hoc* tactical networks. The goals of this project are to determine the feasibility of this concept, to uncover the necessary engineering principles, and to build and experiment with a prototype. This work is funded by the Air Force Research Laboratory, Operational Information Management group.

The notional system underlying this research, called *HotDiffusion*, is intended to function under the very challenging network conditions prevalent in *ad hoc* tactical networks: highly dynamic, unpredictable, and with low bandwidth. The problem is to provide useful information management (IM) services in this setting; specifically, some means for creators of information to publish it so that it can then be discovered by, and delivered to, other clients of the system.

The conventional approach to IM for enterprises is based on a client-server model. The very poor performance of routing in ad hoc networks that experience frequent partitioning suggests that such an approach will not work well in tactical networks. Even dramatic improvements in ad hoc routing will not make the client-server approach attractive in tactical networks.

A peer-to-peer architecture for IM, in which the nodes of the ad hoc network provide services to one another, is a promising alternative. At a minimum such a system would entail the persistence of published information at one or more nodes, some means for a client of the system to discover a copy of desired information, and the routing of the information to the client. To provide reasonable performance in the face of network partitions, replication of published information is essential.

The design of such a peer-to-peer IM system must confront the problem of where to place the replicas of published information – informally, to *cache* the information – especially in light of limited resources. An attempt to place the information in strategically located caches based on policy, heuristics, or learned patterns could be very effective. However, these mechanisms can be expensive to operate. When the assumptions underlying a policy or heuristic are violated, the resulting cache placement may further degrade performance.

In a similar vein, such systems can also employ *advertisements* – messages which allow nodes to build up indices of information cached at other distant nodes. Such indices can be very powerful when they are fresh, and point to information sources that are actually accessible. However, they can also be expensive and brittle.

HotDiffusion avoids both of these problems. Information is dispersed throughout the network at a desired density by a purely local and dynamic mechanism inspired, in part by diffusion of molecules. No attempt is made to cache information at *specific* nodes. The method uses no global coordination, yet naturally "discovers" and takes advantage of such structures as temporary connections between sub-networks. HotDiffusion does not build remote indices of object caches. Rather, each query for information is satisfied by

the client's own node, and by a small neighborhood of nearby nodes that are reached by a limited radius "flood."

When very small sub-networks (and even single nodes) become isolated, they will carry with them some cached objects, and so will be able to provide at least some service to clients on those nodes – preserving published information, and delivering what is locally available. When such nodes become reconnected to the main body of nodes, they will seamlessly begin to exchange information with them and restore fuller service.

In HotDiffusion, information dispersion occurs via a combination of three mechanisms: *random walk* of information objects between adjacent nodes, *cloning* of objects, and *expiration* of objects. The interplay of these three mechanisms will ideally produce a steady-state distribution of copies of the information object at the desired density. The challenge of HotDiffusion is to design and validate a purely local control scheme which will cause the object population to converge to the desired distribution.

This research effort has focused on three areas or goals:

1. To invent and design a control scheme for object dispersion. To understand its expected performance using a combination of simulation and analysis.
2. To build a prototype of HotDiffusion. To test this using emulation of the ad hoc wireless communications.
3. To build a testbed of mobile wireless nodes to host the HotDiffusion prototype. To conduct an exercise with the testbed illustrating HotDiffusion capabilities.

The remainder of this report is structured as follows: Section 3, *Methods, Assumptions, and Procedures*, begins with a description of our assumptions about the environment in which HotDiffusion functions. It also discusses our approach to simulation and network emulation, including some limitations that are implicit in them. The host platform for HotDiffusion provides, primarily, some communication services, which are discussed next, including their implementation and emulation. The following subsection discusses measures of performance – objective qualities for which our design for HotDiffusion strives. This section continues with a detailed explanation of the HotDiffusion algorithms, including their final implemented form. The section next describes the HotDiffusion system architecture. Following that is a description of our wireless testbed nodes. The section concludes with a summary of our experimental plan.

Section 4, *Results and Discussion*, begins with results in three main areas: 1) the crucial realization that density is the primary determinant of availability; 2) results of the Marker Trail method of measuring object density; and 3) some initial results concerning the application of feedback control to the cloning and expiration of objects. Following this we describe results of comparisons between the simulation of HotDiffusion, our prototype implementation, and two other information management architectures.

Section 5, *Conclusions*, recaps our findings and Section 6 summarizes our Recommendations.

3. Methods, Assumptions, and Procedures

3.1 Assumptions about Tactical Networks

HotDiffusion provides information services to clients in the very challenging conditions of an *ad hoc* tactical network. We have made assumptions about the nodes and links that are consistent with this target environment.

Links between adjacent nodes in such networks can have low bandwidth. They may also have highly variable connectivity, with links forming and breaking without warning. Links may be sparse enough that network partitions are common, and isolated nodes are also common. Under these conditions conventional ad hoc networking will perform very poorly. Necessary routes between pairs of nodes may not exist. If such routes do exist, they may only persist for a short while, and thus the cost (effort) of discovering them and repairing them may not be worthwhile.

We have assumed a possibly sparse and dynamic link-level *ad hoc* network – with no services at the network layer (layer 3) or higher. Thus, there is no multi-hop packet routing and no transport assumed. Each link is a one-directional unreliable connection between two nodes, such as might be possible between two adjacent elements of the tactical network that are in direct radio contact. Across these links each packet is either delivered uncorrupted, or it is not delivered. A link that was subject to bit-level errors (as well as possibly missing packets) could be easily adapted to provide this service by application of an appropriate checksum. We will assume that this has been done.

Network topology and link quality are assumed to be highly dynamic. Link quality is characterized by the rate of packet "loss" on the link, which includes packets that have been dropped because of corruption. Packet loss is assumed to be uncorrelated – an assumption that may not be fully justified, given our target environment.

Some of the nodes in a tactical network will be devices with relatively limited capability for computing and data storage. While we have not made a quantifiable assumption about this, we feel that the HotDiffusion algorithms we intend to implement will be suitable for very modest hosts.

Almost every device in a tactical network will be equipped with Global Positioning System (GPS). We assume that every HotDiffusion node has this, and that the position is continuously available to our software. Although the basic HotDiffusion algorithms do not refer to geographic position, we envision creating information densities that are dependent on a variety of external factors, including position. Position is a particularly good example of such an external factor – one that almost certainly will be of interest in any deployment situation, and relatively easy to experiment with.

3.2 Network Simulation and Emulation

We have modeled links as having piecewise-constant packet loss rates. Our emulator is capable of representing packet loss rates in the full range [0.0, 1.0]. However, most of our experiments and reported results use a simpler model where each link is either *on* or *off* (packet loss rates of 0.0 or 1.0 respectively). Where we have simulated or emulated mid-range packet loss rates, the lost packets are uncorrelated.

For most of our work, we have used a *random-waypoint geometric graph* (RWGG) model. In this model, nodes move along straight lines between uniformly distributed random waypoints, with randomly chosen speeds. The waypoints all lie within a unit square. Whenever two nodes are within a given *connection radius* they are connected in both directions (having a link quality of 1.0 or a packet loss rate of 0.0). Nodes that are separated by more than the connection radius cannot communicate. By varying the connection radius for a fixed number of nodes we can model networks with different character.

In the RWGG model node motion is not correlated. However, nodes are more likely to be found near the center of the square than at the edges. Also, the links between nodes are somewhat correlated: Nodes that are connected to one another are also more likely *both* form a link to some third node. The independent motion of nodes will produce connections between pairs that form and break in complex combinations. It also produces a kind of thorough mixing of nodes – over long time scales all pairs of nodes are equally likely to form links. This even mixing is somewhat unrealistic, in our understanding. Are these idealized conditions more challenging than real correlated motions? We believe that the arbitrary nature of the links (and connected components) that form in such a network, with no potential for long-term prediction, will pose problems for some information flow schemes, such as those that take a predictive, knowledge-based approach to caching information. In contrast, HotDiffusion takes advantage of node mixing both to enhance stability and for information transport. Thus, our simulated networks contain this feature (non-predictive mixing) which is certainly present to some degree in tactical networks, and that HotDiffusion will opportunistically take advantage of when it is present.

In our use of the RWGG model we select random node velocities uniformly in a range from a smaller value up to a maximum value that is ten times greater. This distribution of speeds, in combination with the given connection radius produces a distribution of lifetimes for links. A faster moving node will typically remain connected to other nodes for shorter periods of time[1]. However, in our model, every node gets a new random speed at the next waypoint, and so has a chance to form more long-lived connections. The typical connection lifetime for a node is important for HotDiffusion because it must

[1] Slow moving nodes that have a "grazing" incidence in their motion will have a short-lived connection. However, a node with high speed will only have a short-lived connection with almost all nodes encountered, except another high speed node that is moving roughly parallel to it – a relatively rare occurrence.

be longer than two other timescales in our system: the time to recognize a "good neighbor" and the time between object migrations. *In our current implementation*, the parameters associated with the rates of these mechanisms are constants that would need to be adjusted so that these timescale inequalities hold. Our model is thus limited to situations with a uniform node behavior with a constant distribution.

In both our simulation and emulation we have modeled the packet communications without latency and without congestion. The time for a packet to be sent and received is so much shorter than the other timescales in the system that the effect of latency is unlikely to be a major factor. Congestion could be an issue in a real deployment of HotDiffusion. However, depending on the link technologies in use, congestion should be well approximated by a modest increase in latency or packet loss.

3.3 Design Elements of Host Platform

3.3.1 Communication Library

As explained above, HotDiffusion is intended for applications where unreliable link-level communications between adjacent nodes are the norm. HotDiffusion makes use of this capability, and does not assume or make use of any "higher" network functions such as routing. Thus, the basic functions assumed are to broadcast a packet, or to send it to a specific adjacent node, and to request an incoming packet if there is one. Incoming packets are assumed to be buffered and check-summed. We have encapsulated this functionality in a C language API as follows:

```
hd_status hd_initialize();

hd_status hd_send(const hd_msg_ptr msg, const hd_name_ptr dest);

hd_status hd_recv(hd_msg_ptr msg, hd_name_ptr from);

hd_status hd_getname(hd_name_ptr name);

int hd_comparenames(const hd_name_ptr, const hd_name_ptr);

hd_status hd_shutdown();
```

The `hd_initialize()` function is called before using the interface. The `hd_shutdown()` function is called to indicate that the interface will not be needed. To send a packet we use the `hd_send()` function, whose `dest` parameter indicates either a specific recipient, or a broadcast. The `hd_receive()` function retrieves an arrived packet if there is one, or indicates via its status that there is none. The node can learn its own network name via the `hd_getname()` function. In this API, the node names are arbitrary binary data.

We implemented this API on top of UDP and IP. The implementation uses, in effect, the four bytes of IP address as the node name. Depending on a configuration parameter, the implementation will operate either in a "direct" mode or an "emulation" mode.

Direct Mode

In the direct mode, each outgoing packet or message is encapsulated in a UDP packet and sent to the indicated IP address, at a fixed port number. The `hd_recv`() function then receives UDP packets at that port. The UDP destination could be the designated IP broadcast address for the network. This implementation has several advantages:

- If the platform is equipped with a wireless interface card, and that card is placed in ad hoc mode, then the behavior is as desired: Broadcast messages are received by all nodes that are within range of the sender. Point-to-point messages are received by the designated recipient, if they are within range. Message delivery is unreliable. No multi-hop routing occurs.

- If the platform is on an Ethernet LAN, then it will appear to be "adjacent" to all other nodes that are running on that LAN with respect to both broadcast and point-to-point messages.

- A host computer may be configured to use multiple IP addresses on one interface (e.g., one Ethernet interface). Then multiple instances of this software may be run on that host, each one configured to use a distinct IP address. All of these instances appear to be "adjacent" to one another, and adjacent to all other instances on the LAN. This allows the software to be tested in relatively large configurations with only a few hosts.

- The library implementing our API can be implemented entirely in the "user space" of a Unix host, greatly simplifying development and testing.

Emulation Mode

In the emulated mode, our implementation still wraps each outgoing message in a UDP packet. However, this packet is now sent to a running instance of the ABSNE emulator. There, it is unwrapped, and ABSNE compares the (source, destination) pair are compared with a time-evolving model of connectivity. Using the link quality between that pair (in that direction) at that moment, the emulator decides pseudo-randomly whether to deliver or drop the message. If the message is to be delivered, then it is rewrapped in a UDP packet, and sent to the recipient.

In the case of a broadcast message, the emulator probabilistically forwards the message to all of the potential recipients via multicast. It makes an independent decision about forwarding to each recipient.

Emulation works in conjunction with multiple IP addresses per network interface. In this way a large emulated network of nodes can run on one or a small number of hosts. The ABSNE emulator typically runs on a different host.

3.3.2 Neighbors

HotDiffusion requires reliable bi-directional communications on each usable node-to-node link. We also require that nodes are able to address their immediate neighbors unambiguously – they must know the identity of the node at the other end of each link. We have implemented this capability via a combination of node discovery and link quality estimation.

Each node broadcasts a beacon message containing its identity, on a regular schedule. Any node that receives such a beacon message may begin to accumulate link quality statistics for that link.

We are using a sliding window link quality estimator. If node-A has received K out of the last W beacon messages from node-B, then the link quality from A to B is estimated to be K/W. Since the beacons are broadcast on a regular schedule, the recipient knows if it is missing any.

Nodes will only use links that are "good" in both directions for object migration, because they want to be reasonably certain that the object will arrive, and that an ACK message can be returned. Link quality is deemed good if it exceeds a fixed threshold.

How does a node know that its outgoing link quality to some other node is good? The beacon messages contain a list of the nodes whose incoming link quality is better than the threshold. Thus any recipient of such a beacon can determine when link quality is greater than the threshold in both directions by verifying that its node ID is included in the list sent with the beacon.

3.4 Performance Measures

HotDiffusion offers an information management service to programs running on tactical nodes. With this service, client programs can publish information objects which the system persists and makes available to other clients. Clients may query for objects matching a pattern that they specify. There are many reasonable measures of success for the service that HotDiffusion offers. We focus these issues: When an object is published, does the system preserve and disperse it? Can nodes retrieve published objects that match their queries? How certain is this? How long does it take?

3.4.1 Controllability or Stability

Ultimately, HotDiffusion functions by dispersing copies of published objects among the nodes. The goal of this dispersal is both to place copies near (or at) nodes that may query for them, and to provide redundancy so that objects are not lost. Although making enough copies quickly and spreading them around is good, making too many copies is not good, since this consumes resources. Thus, the issue is whether the density of object

replicas stabilizes quickly and whether this final density can be controlled over a useful range. Since we have been using networks with a fixed number of nodes, we are primarily interested in the total number of object replicas as a function of time.

In some earlier versions of HotDiffusion, fluctuations in the number of replicas, in combination with some phenomena that occur in small isolated subnets, had the effect of causing published objects to "go extinct" – for the number of replicas of the object to go to zero. We have since enhanced the HotDiffusion algorithms (and implementation) so that extinction will never occur in a network with a fixed set of nodes. However, in situations where nodes may be "lost," if there are only a few replicas of an object, it could become extinct. When the number of object replicas expands promptly to the desired level and remains stable, the system will resist object extinction via node loss.

3.4.2 Availability

HotDiffusion queries are satisfied by looking for matching objects at the query node, and in the nodes nearby. It may be that a published object that matches the query is still not found among this small set of nodes. In that case we say that the object is not available to that client. *Availability* is the fraction of matching published objects that are returned to the client.

3.4.3 Latency

If a query fails to find a matching object, it may be appropriate to repeat the query at intervals in order to take advantage of the network dynamics and circulation of objects. We may thus ask what the availability of an object is (or would be) given that the client is willing to wait and retry up to some specified time interval, the *latency*. The natural expression of this trade off is a graph of availability vs. latency.

Thinking of availability and latency in this way also provides a more natural (and fairer) comparison of HotDiffusion with some competing architectures such as Disruption Tolerant Networking, in which responses may take an unbounded length of time.

3.5 HotDiffusion Algorithms and Implementation

The HotDiffusion Core is a program that runs on each mobile ad hoc node. It handles object dispersion and provides information management services to clients running on the node. In this section we describe, in detail, how this program works.

The functionality of the HotDiffusion Core can be divided into three major areas: (1) estimating the local density of information object replicas, (2) managing object migration, replication, and expiration, and (3) handling the satisfaction of queries. These fit together roughly this way: The density of replicas for each distinct published information object is estimated continuously in the neighborhood of each replica. This estimate, compared with a target density, is used to control the fate of that replica – to replicate (or "clone"), expire, or simply migrate. Ideally, replicas of a published object will thereby expand rapidly throughout the network, and stabilize at the target density

(per node). Finally, in response to a query, the core looks in its local cache of objects, and among its neighbors for matching objects.

3.5.1 Estimating Density – Marker Trail

For each object that a node holds, it needs an estimate of the density of replicas of that object (i.e., object replicas per node). The estimate is used by the node to determine the next step for the object. The estimate should track the evolving population of replicas. It should be sensitive to local variations in density so that it can respond to local fluctuations. Yet, it should not be too local since, at the scale of single nodes, any node that holds a copy of an object is a "hot spot" for that object. When an object replicates (as described in the next section) we end up with two copies of the object at adjacent nodes – another small-scale hot spot for the object.

Our method of computing this estimate is called *Marker Trail (MT).* Each object carries with it an evolving estimate of the density in the region it has recently visited. It leaves behind a "trail" – a small amount of data – at those recent nodes, and examines the trails left by other clones of the same object. Thus, in considering the next action for an object, the node will use the density estimate that the object carries with it.

Here is the basic concept of MT: Objects are assumed to move between adjacent nodes at regular intervals called a *time step*. At each step the object leaves behind a "marker" containing the object's ID – the name that is shared by all clones of the same ancestor object. These markers are designed to persist at the node for a certain fixed time, and then expire (i.e., disappear). If the target density for replicas of the object is 1/K, then markers will persist for K time steps. With this setup, if there is an object with a stable population of replicas at the target density in a network of N nodes, then there will be on average, one marker per node.

N = number of nodes

K = time-to-live for markers, in units of the time step

= length of each object's trail at any moment

N/K = desired number of replicas of the object

(N/K) K = N = total number of markers

As the object migrates from node to node, it keeps track of how many markers containing its own ID are encountered. (Of course, software in the node implements this.) The *basic* density estimate for the object is the average number of markers encountered per node visited divided by K. Thus, the target density of objects corresponds to one marker per node in this estimate.

In order to maintain a running estimate, each node is created with an estimate that is initialized to 1.0. As each node is visited, the estimate is adjusted, using a *decay factor*, *d*, less than 1. This is how the estimate, E, is updated at each step:

$$E_{n+1} = E_n\, d + \langle \textit{number of markers} \rangle\, (1 - d)$$

By adjusting the decay factor, the number of hops that the object "looks back" along its path can be modulated. This measure counts more recently encountered markers more strongly, which seems sensible.

It turns out that the basic MT, as described here has several weaknesses. We have designed compensating mechanisms for two of these (uneven object distribution and self-trail interaction), as described below.

Uneven Object Distribution

Objects that random-walk uniformly on a fixed *connected* network of nodes (i.e., a fixed set of nodes with fixed connections and no isolated nodes) where at each step the next node is chosen randomly with equal probability among the neighbors, will be found at nodes of high degree[2] more often than at nodes of low degree. If the object has been random-walking for a "long" time, then the probability of finding it at a node is proportional to the degree of that node.

Migrating objects in a stationary distribution will lay down trail markers as the object visits a node; the marker is created and is left behind at the node as the object migrates to another node. Markers consist of three values: the global ID, which represents the ID of the original object, the local ID of the instance (clone) that is at that node, and a time to live field. This has a low impact on resource constrained nodes and markers persist for a limited time.

Markers will have the same density distribution (up to a constant factor) as the objects themselves. So, if the objects favor some nodes over others, the trail markers will also be more concentrated at those favored nodes. In parallel with this, the migrating objects are "looking" for markers, and will be more exposed to those markers that occur at the favored nodes. The net result is an overestimate of the density.

Let P_i be the *probability density* of an object in a stationary distribution resulting from random walking. That is, P_i is the expected number of copies of the object at node i. If there are A copies of the object in circulation, then the sum over i of P_i will be A. If there are K trail markers per object, then the expected number of trail markers at node i will be $K P_i$. In light of this, we can express the expected number of markers per node seen by a migrating object this way:

$$Q = \sum_i (P_i/A)\,(P_i\, K) = (K/A) \sum_i P_i^2 \quad (1)$$

The minimum of this expression for fixed normalization occurs when all of the P_is are equal. We can express the P_is this way:

---

[2] The degree of a node is the number of immediate neighbors it has. In graph theory, the degree of a vertex in an undirected graph is the number of edges incident on the node, with loops counted twice.

$P_i = A/N + D_i$ where $\sum_i D_i = 0$ *(2)*

Substituting into the expression for Q gives:

$Q = K\ A/N + K/A \sum_i D_i^2$ *(3)*

The first term in (3) is just the intuitive answer when all probabilities are equal: The number of markers divided by the number of nodes. The second term is non-negative, and is zero only when all of the P_is are equal.

We considered methods for selecting the next node that would even out the probabilities. However, it turns out that this *necessarily* involves uneven object motion – the object must "rest" at some nodes longer than others. This uneven motion would greatly complicate the analysis of the trail. In the end, we did not do this.

In dynamic networks, where all nodes are equivalent, this effect is washed out in the long run. We are concerned with effects that can manifest in timescales of roughly

T = timeStep / (1 – decayFactor)

We have been experimenting in a regime where there is sufficient dynamism that this effect is small. It could become important in fixed ad hoc networks or those with slowly evolving connections. However, in those settings, there may be alternatives to HotDiffusion that are preferable.

Self-Trail Interaction

As objects migrate from node to node, they remain near[3] their last few positions and thus, near their own trail markers. As a consequence, they are likely to "cross their own trail." They are more likely to see their own trail markers than those of their clones. If they count all trail markers containing their parent object ID, then this effect will cause a tendency to overestimate the density. The effect is generally greater in networks with sparser connections.

To correct for this effect we introduce two IDs for each object. The global ID (GID) is given to the object when it is published, and is inherited by all clones. The local ID (LID) is distinct for each clone. Both GID and LID are included in trail markers. In accumulating a marker density estimate, objects will ignore their own trail, using the LID for this purpose. If done naively, the estimate would be too low. For a target density of Δ, we would use a trail length $K = 1/\Delta$. In a network with N nodes, if the objects are at the target density, then there will be $N\Delta$ objects. Our naïve self-ignoring object will see a density of markers:

Marker density = $1 - 1/(N\ \Delta)$.

---

[3] In this context, the term *near* means a small number of hops in the network graph.

This apparent density is different from one; a bit too small. We compensate for this by adjusting the trail length:

$K = 1 / (\Delta - 1/N)$

With this longer trail, objects at the target density will see an apparent marker density of one, ignoring their own markers.

The one subtle point about this correction is that objects (or nodes) must know or estimate the size of their network. In some circumstances this number might not be known. This value for K should produce long-term average density estimates that agree with the global average density.

Some networks may contain nodes that are temporarily disconnected from all others. Object replicas at such nodes cannot move. Yet, our model is that objects move and leave a trail marker at each time step. In our implementation we chose to address this situation as follows: When an object is at an isolated node, and its "time step" is done, then it simply waits until a connection to some other node forms. Soon after that, it moves to the newly connected node (or replicate, or expire). It only creates trail markers when it can move. These markers are left behind as the object moves on. The object only updates its density estimate when it moves, using the markers in the node to which it has come.

This method of handling objects at temporarily isolated nodes distorts the density estimate somewhat. The trail of an object that has been delayed in this way will be shorter, thus causing the density estimate to be too low. We considered allowing such objects to put down markers and update their estimates, which would maintain the global statistics on number of markers. Unfortunately, this would also cause a persistent clump of markers that would cause the density estimates to be much noisier.

An alternative approach to compensating for delayed objects is to increase the lifetimes of the markers that are eventually created, depending on the length of delay. This seems entirely feasible – the momentary deficit of markers is followed by a temporary surplus so that the long term average number of markers is correct. We did not implement this.

3.5.2 Controlling Density – Cloning, Expiration, and Extinction

Each object maintains a running estimate of the density of its replicas in nodes that it has visited recently. If this estimate is too small (smaller than the target), then the object may clone or replicate. If this estimate is too large, then the object may expire. This simple strategy is intended to keep the population of replicas near the target density. It also serves to level-out clumps and depressions in the distribution that may develop. The details of how this is implemented turn out to be important.

Initially we used the error signal (the difference between the density estimate and one) multiplied by a feedback parameter as a probability. If the result was positive, then that was a probability that the object would expire. If the result was negative, then it is

interpreted[4] as a probability of cloning. Although this worked to some degree, it exhibited a number of problems. The population numbers exhibited overshoot. There was some potential for instability, which could be countered by a very small feedback. Small feedback, in turn, resulted in a slow growth of the object replica population. Fluctuations in population could cause local and global extinction of objects. Extinction was a particular problem with very small isolated subnets.

Our final version worked like this:

1. If the node is isolated, then wait.

2. If the node has neighbors, then the object with the earliest expired timer (if any) is considered. Thus, if objects have been delayed at an isolated node, the one waiting longest is handled first.

3. If there are multiple copies of this object at the node, then the average of their density estimates is computed. If this average is greater than a threshold (typically set at 1.0), then the object (replica) expires. It is simply removed from the system.

4. The error signal is computed as described above. If this is negative, then it is multiplied by a feedback parameter, and is used as a probability of cloning.

5. If the object is not expired and not being cloned, then it will migrate.

6. In either case (cloning or migration) a neighboring node is chosen randomly with equal probability, and the object is transmitted to that node. The object carries with it an estimate of density. If it is being cloned, the estimate is set to 1.0 in the transmitted copy. Otherwise the existing estimate is used.

7. A timer is set, to define an interval of time for an acknowledgement.

8. When the acknowledgement is received, if the object is migrating, then the local copy is deleted from the cache. If the object is being cloned, then density estimate is set to 1.0 in the local copy, and the object is assigned a new LID.

9. If the timer expires without receiving an acknowledgement, then we try again, including re-evaluation of the object's fate and possibly selection of a new random neighboring node.

When an object clones, the two nearby copies will likely constitute a considerable "bump" in the distribution. This very localized high density will gradually smooth itself out over time. However, we do not want these objects to sense this density bump too soon, and needlessly expire. So, we place the original and cloned objects at different nodes, so that they will more likely have divergent histories and will move apart. For

[4] The probability is the absolute value of this quantity.

both original and clone, we also reset the density estimate to 1.0 – a neutral value, to give them an opportunity to move apart and to re-evaluate the local density.

In very small subnets (e.g., two connected nodes, otherwise isolated) even a single copy of an object may exceed the target density. With ***the original control scheme***, the object would quickly achieve an accurate density estimate and then expire! In some sense this is the correct behavior, but not desirable. Switching to a method in which density is measured by counting only markers of other clones, reverses the problem: The isolated object's density estimate plunges to near zero, at which point the object clones. Now, with two objects batting around in the subnet, each one sees the markers of the other, and again the density estimate soars. Once the estimate is high, there is a good chance that both objects will expire. If only one of the clones expires, then the other one will repeat the "roller coaster" ride.

With our new method, described in step 3 above, objects only expire when they meet at a node. In such collisions, if the average density estimate of the colliding objects is greater than a configurable threshold, then one of the objects is eliminated. In this way, isolated objects in small subnets cannot go extinct.[5] Such objects will still experience the sequence of declining density estimates, cloning, and eventual expiration. However, we are guaranteed that only one of the objects will expire, so the sequence can repeat forever.[6]

Some "jitter" is introduced into the time steps so that object motions are not synchronized.

3.5.3 Querying – Flooding Requests

Clients issue a query that contains a predicate, as well as a desired number of matching answers by sending this to the HotDiffusion Core running locally on their host node. The Core evaluates that predicate against all of the objects it currently holds. Any objects for which the predicate evaluates to true (up to the number requested) will be added to a queue of responses for this query. If multiple copies (clones) of the same object reside at the node, then only one of these will be added to the queue.

If the responses gathered from the node's object cache are fewer than the client's request, the query will be propagated to neighboring nodes, seeking the remaining number. This is done via a broadcast message.

In the fullest conception of HotDiffusion, these query messages would flood out via re-broadcast for a fixed small number of hops. Each node that re-broadcasts will decrement

---

[5] This assumes that the population of nodes is fixed. If nodes can "disappear" then the objects that they hold can become extinct if they are not, at that moment, well replicated.

[6] Ideally, small isolated subnets do not remain isolated forever – they eventually mix back in to the general population of nodes.

a time-to-live counter, and will add itself to the sequence of nodes in a "route" field. Each recipient of such a message also applies the predicate to all objects that it holds, and sends any matches (up to the number requested) back along the accumulated route. The assumption is that such routes usually remain valid for the few seconds necessary to forward and process requests.

Several simple enhancements will improve the efficiency of this scheme: Each query contains an ID. Nodes will only forward and respond to a query message once, typically the "version" with the shortest route. Nodes will, of course, only respond with unique matching objects. Since they are forwarding objects for nodes farther out, they can filter (i.e., not forward) responses that have already been sent via that node.

Our implementation of the HotDiffusion Core implements only one-hop queries. That is, queries are satisfied by the local node, and by those directly reachable via a broadcast message. The decision to limit our implementation in this way was made because at the scale and density of networks we anticipated using in our experiments (as opposed to simulating), we would probably want to select a query radius of one hop. This one-hop query mechanism is much simpler to implement and is equivalent to the multi-hop case where the radius is set to 1.

3.5.4 Architecture

The HotDiffusion Core is a program that runs on each mobile ad hoc node. It interacts with client programs also running on the node (if any), providing information management services. The Core also interacts with other nodes via the Communications Library. Core-to-Core messages support node discovery, object migration, and querying. Figure 1 shows the architecture of the HotDiffusion system.

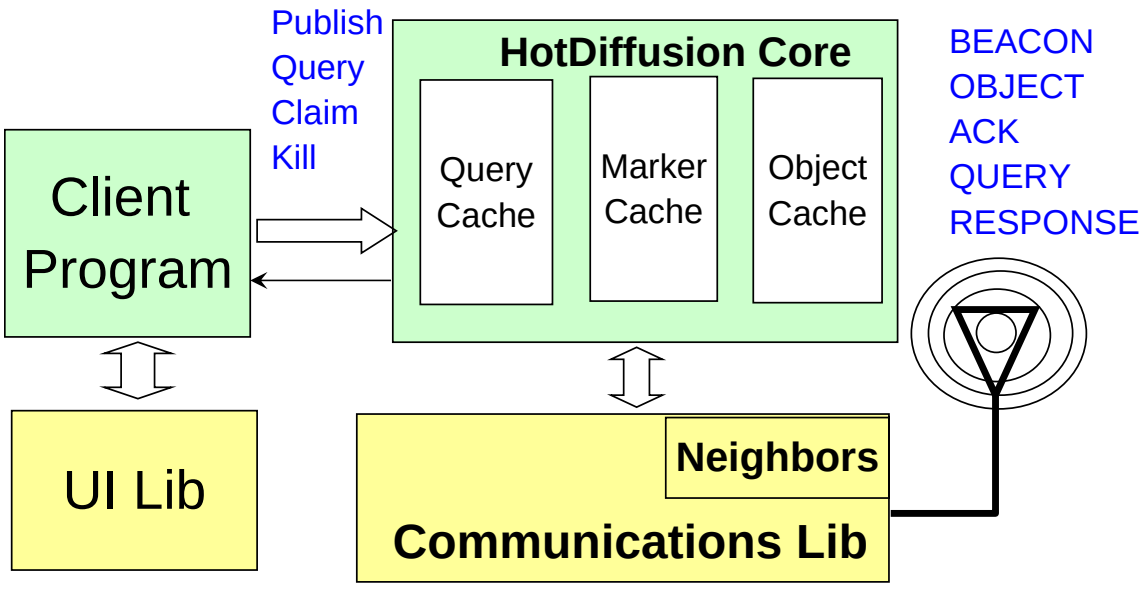

Figure 1. HotDiffusion Architecture

3.5.4.1 Object Model

Information objects in HotDiffusion are sets of key/value pairs. Each key and each value are null-terminated strings. Some of these pairs are used by the HotDiffusion infrastructure to carry state and parameters along with each object. For example, each object carries a target density and a running estimate of density. These pairs are assigned by HotDiffusion and are not necessarily constant. They are visible to clients who receive the object, if they are interested. Two very important key/value pairs are "GID" and "LID". The values associated with these keys are globally unique identifiers assigned by HotDiffusion.

Other key/value pairs are assigned by the client program to carry application-relevant data. From the point of view of traditional information management, we may think of these pairs as a combination of metadata and payload.

Predicates over information objects play an important role in queries. We have developed a simple language for expressing queries over objects that are structured as arbitrary key/value pairs. Predicates are either primitive – referring to a specific key/value pair – or are Boolean combinations of other predicates. The main novel feature of this system is that primitive predicates must explicitly state how they are to be treated if the named key is not present (either as *true* or *false*).

Primitive predicates may treat the value associated with a key as either an integer or a string. For example, we may require objects where "size" is less than 10. We would write this as:

LT(!size, 10)

Or, we might seek an object whose "name" is exactly "object_seven" this way:

EQSTR(?name, 'object_seven')

Here, the ! in front of *size* means that this predicate evaluates to *false* if the key *size* is not found. The ? in front of *name* means that the predicate evaluates to *true* if the key *name* is not found. A compound query which is the conjunction of these two would be written this way:

AND(LT(!size, 10), EQSTR(?name, 'object_seven'))

This system could easily be extended to encompass other data types (e.g. floats and binary blobs) and other operators. Arithmetic and other predicates involving multiple keys would also be straightforward to implement.

3.5.4.2 Client Operations

Clients interact with HotDiffusion by sending one of four kinds of messages to the Core on their local host:

1.Publish

2. Query

3. Claim

4. Kill

In our prototype implementation these messages are delivered via TCP/IP sockets, using the local loopback device. There is a separate thread in the Core associated with each client that responds to messages from that client. Locks within the Core software mediate contention for shared data structures. The HotDiffusion Client API, supporting connecting, publishing, and querying, is available as a C library.

Clients may construct a local object as a data structure of type *metadata* which contains the key/value pairs. They may use a variety of tools to manipulate objects, and may locally modify existing objects received from HotDiffusion. To publish an object, they send a Publish message to HotDiffusion containing the object, the desired density per node of replicas of this object, and an optional lifetime. The HotDiffusion Core fills in (or overwrites) all of the key/value pairs that are part of the infrastructure, including GID and LID, and adds this object to the local object cache. The lifetime of an object is a time interval in seconds, after which all copies of this object will be deleted.

To query for information, the client formulates a predicate using the notation described in Section 3.5.4.1. This string is passed to the Core together with a desired number of matching objects in a Query message. The client receives back a *claim ticket* which will be used to claim the matching objects. To claim objects, one at a time, the client sends a Claim message to the Core. If the core has found one or more matching objects that are unclaimed, it responds with one of them, and with the number of additional unclaimed matching objects that it *currently* has. If the client has already claimed objects up to the requested number, or has *killed* the query, then the query is *done*. From the Core, the client may receive an indication that either there are no matching objects, or that the query is done.

3.5.4.3 Core Operations

The HotDiffusion Core receives and responds to four kinds of messages from clients (Publish, Query, Claim, Kill). The Core also exchanges five kinds of messages with its peers on other nodes:

1. BEACON

2. OBJECT

3. ACKNOWLEDGEMENT

4. QUERY

5. RESPONSE

When the client publishes an object (via the Publish message), the Core will add or overwrite key/value pairs that are required for its own functioning, including GID and LID. It then adds this object to its local *object cache*, where it begins the process of diffusion, and is available to match queries.

When the client initiates a query (via the Query message), the Core compiles the predicate and creates an entry for it in the *query cache*. The compiled predicate is evaluated against all objects in the object cache. Any *unique* matches (up to the requested number of matches) are added to the entry in the query cache. Here, uniqueness is determined via the GID value only. If the requested number of matching objects is not satisfied, then a QUERY message is sent via UDP broadcast to all neighboring nodes requesting the remainder. The QUERY message contains a unique identifier for the query.

Each recipient of a QUERY message looks in its object cache for the requested number of matches. If any are found, each unique one is sent back to the query node in a RESPONSE message containing the object and the unique query id. Note that this mechanism only implements one-hop queries. Multi-hop queries would require that nodes rebroadcast QUERY messages, accumulating a return route. Responses would need to follow that route back to the query node.

When a node receives a RESPONSE message, it finds the corresponding query cache entry (using the query ID) if it still exists. If the response object is not already in the cache (unique up to GID), and if more matches are still needed to complete the query, then the matching object is enqueued in the entry.

The HotDiffusion Core broadcasts BEACON messages on a regular schedule. It uses the receipt of these to discover its neighbors and to measure the quality of the links to its neighbors, as described in Section 3.3.2.

The Core uses the OBJECT message to carry an object to a neighboring node, when the object is migrating or replicating (cloning). The ACK message tells the recipient that the OBJECT has been received. The handling of these messages is described in Section 3.5.2. When an object has left a node (in case of migration or replication), then it leaves behind a *marker* – an entry in the *marker cache*. Each such entry is created with an expiration time in the future. When objects arrive at a node, they update their density estimate, scanning all of the markers in the cache. Any markers whose expiration time has passed are deleted and do not contribute to the estimate.

The prototype implementation of HotDiffusion requires each migrating object to fit into a single OBJECT message. This, in turn must fit into a single UDP packet, in our implementation. This limits each object to about 65,000 bytes of data. However, we have not tested the implementation with large objects approaching that size.

Aside: Our intended target platforms for HotDiffusion are relatively primitive systems that may not have UDP or its equivalent. Three important services that UDP provides (over *ad hoc* wireless) that we rely on are: addressing transmissions to specific port and recipient as well as broadcast, packet fragmentation and reassembly, and check summing. Porting HotDiffusion to such platforms would likely require finding or implementing substitutes for these.

3.6 Wireless Testbed

We constructed a network of small mobile computers with wireless capability in order to demonstrate and test our implementation of HotDiffusion. We describe here the major features of the nodes of this network.

Each node of the testbed contains a Gumstix computer. This is a single-board computer, about 1.5 cm by 8 cm, which runs a version of the Linux operating system. This is paired with two other boards of similar size from the same manufacturer: An 802.11g wireless board and a GPS board which also serves as a breakout board. The wireless board supports the full TCP/IP protocol stack in the Gumstix. The ssh remote shell program can be used to log into the node over the wireless network to have command line access the Gumstix to install, configure, and operate software. Software for the Gumstix is written in C or C++ and cross-compiled on a Linux desktop computer using the Gnu gcc compiler.

Each node has a 2 line by 24 character LCD display, 5 pushbuttons, and 3 LEDs. These are connected to the processor via the GPS breakout board. Three C-cell batteries power the unit, connected via and on/off switch. This is all enclosed in a clear plastic case, roughly 10.5 cm by 16 cm in area and 5 cm deep. Figure 2 shows the front view of a testbed node with 802.11g antenna protruding from the top. The LCD, visible through the plastic cover, is surrounded by four buttons, with three LEDs at the top. A fifth button (red) is to the left. Figure 3 is an inside view of the node. The GPS antenna is just above the three batteries in this picture. The Gumstix board and the two accessory boards are in a tight stack to the left of the batteries. The PC boards support the LCD display, buttons, and LEDs and are seen from the back in the lower half of this figure.

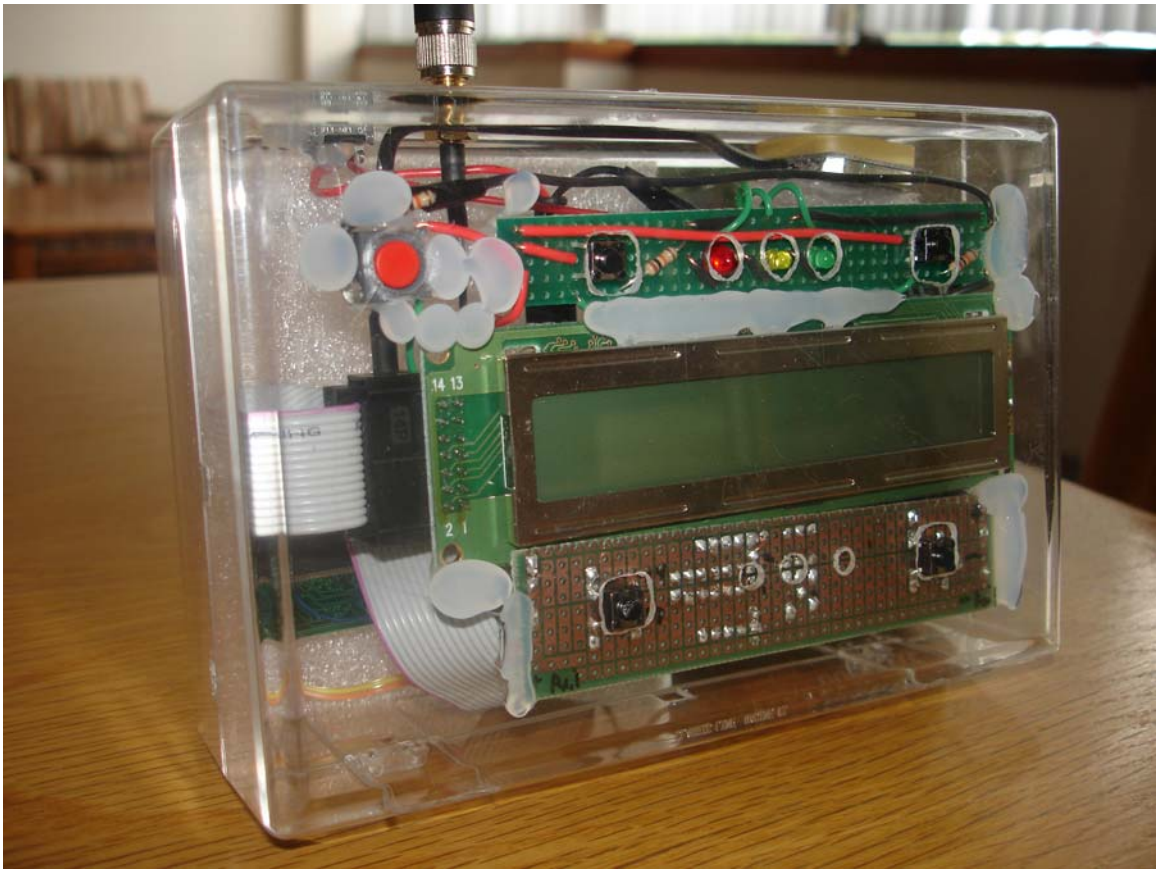

Figure 2. Wireless Testbed Node, front view.

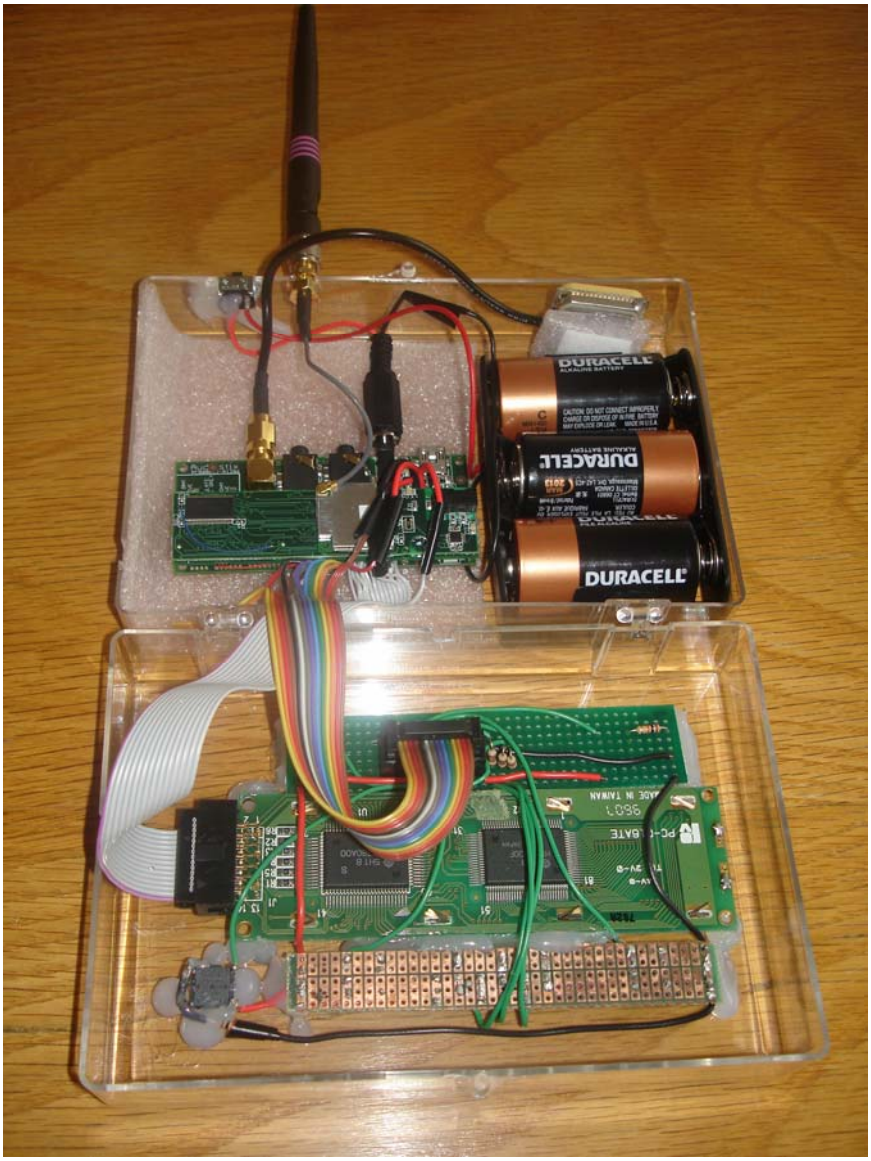

Figure 3. Wireless Testbed Node, inside view.

3.7 Experimental Plan

3.7.1 Goals

The overall goal of our experimental program was to characterize the performance of the HotDiffusion algorithms, and, in particular, to delineate plausible operational circumstances in which they confer measurable benefits. Thus, we measured some quantities, using HotDiffusion, that are associated with useful information management services. We also measured the same quantities in a similar situation, using competing *Information Management (IM)* architectures.

We consider information management services to be useful when they are available, and give timely and accurate answers, and in doing so, consume predictable amounts of resources.

From the client's perspective, information management consists of publishing information (in the form of *information objects*), requesting information matching a given pattern, and receiving the matching information objects. Here are some aspects of this that *clients* may care about:

- **Availability.** What fraction of matching published objects will be returned to the client?
- **Latency.** How long does it take to satisfy a query? There are two separate aspects of this: First, what is the response time for a query? Second, how much time elapses after publication before an object is generally accessible?
- **Availability and Latency combined.** Given that the client was willing to "keep trying," how long (as a distribution) before the object is delivered to the client?
- **Integrity.** Do published objects remain in the system for the specified interval of time? With our improved algorithms, HotDiffusion should never experience "accidental extinction" of objects, assuming a fixed cohort of nodes. However, if nodes are lost or permanently disconnected then it becomes possible. The natural replication associated with HotDiffusion tends to mitigate this.
- **Graceful degradation.** Quality of service falls off gradually as network conditions worsen. When conditions improve, full service resumes.

There are some other properties that clients may care about which are beyond the scope of these experiments, specifically, expressiveness of the query language, and the ability to flexibly manage the query process. In our view, the basic HotDiffusion scheme is compatible with nearly any query language (i.e., a language for selecting individual information objects) that is computationally feasible for the nodes.

Clients are typically not the only stakeholders in the information system. More generally there will be managers or owners, who will be concerned with resource usage, and with the ability to manage the information space. The three primary resources that might be impacted by HotDiffusion and its competitors are processing cycles, node memory, and bandwidth. One of the key issues for HotDiffusion is its ability to rapidly expand the number of replicas of an object, yet not overshoot, and thereby consume memory and bandwidth. The competing architectures are not automatically replicated, and thus do not face this issue at all.

3.7.2 Strategy

To achieve our goals, we collected data over a wide range of scales with varying network conditions and parameter settings. We must also implement one or more competing IM architectures for comparison. These requirements led us to a three part strategy that focuses on simulation.

1. We simulate HotDiffusion and competing IM architectures at scales up to about 100 nodes, running tests that are designed to measure dimensions of performance detailed below. The actual network configurations that arise in the simulations are captured.
2. We use our implementation of HotDiffusion together with the ABSNE emulator to validate the simulation at scales up to about 20 nodes. For selected simulated runs, we use the logged dynamic network configurations as input to drive the ABSNE emulator. We compare the actual HotDiffusion results with the simulated results.
3. Finally, we use HotDiffusion running on our wireless testbed to demonstrate HotDiffusion at scales of about 10 nodes. We run simple exercises designed to create network conditions and publish/query loads that illustrate key performance points as indicated in our simulations. The resulting log files may be analyzed for performance measures.

Thus we can compare real, emulated, and simulated runs.

3.7.3 Competing Information Management Architectures

In order to have a basis for comparing HotDiffusion, we envisioned alternative architectures with idealized performance that is as good as, or better than any real implementation could expect. We simulated the competing information management architectures for direct comparison with the HotDiffusion simulation. We describe two alternatives to HotDiffusion below.

3.7.4 Object Server, Direct Access

In this scheme, published objects are stored at the publisher node or at a specific server node. Clients must contact that node to obtain them. We assume ad hoc routing which works perfectly – if there exists a complete path between the client and server nodes, then the request is successful. The client keeps trying if the object is not accessible so that the request is eventually fulfilled if a path to the server ever develops. In this scheme, we ignore the cost of finding the object – we assume that the client knows which node is the server.

We are primarily concerned with latency – how long before the client receives the object (if at all), and routing cost. We compute a reasonable proxy for routing cost as follows:

3.7.5 Disruption-Tolerant Networking (DTN)

In this scheme, the client must still ultimately contact the server, and the server will reply. However, we assume a store-and-forward style of networking – an idealization of DTN.

We assume that when a request is made, we “mark” all reachable nodes. As the network evolves, we continue to mark all nodes that are reachable from any previously marked node. When the server node is marked, we repeat the process in the other direction, back to the client. We are assuming that DTN works presciently; it magically finds the fastest route to the target node.

We are primarily concerned here with the latency of the request/reply sequence, and with the failure rate given a finite timeout.

3.7.6 Specific Experiments

3.7.6.1 Availability, Latency, and Stability

In this experiment, nodes are moving on a unit square arena under a random waypoint model. Nodes are deemed to be physically "connected" when they are within a fixed *connection radius* of one another, disconnected otherwise. Near the beginning of the experiment, a client at each node (or a subset of nodes) publishes a unique object. Clients at each node (or a subset of nodes) then periodically query for these objects to see if they are available, and if so, what the latency is.

In this experiment, for the two competing architectures, fulfillment of a query will typically take some time, and will require some "retries." For HotDiffusion, we are directly measuring the immediate availability of objects, essentially with zero latency. If an object is not available, we can "look forward in time" to the nearest data point at which the object is available, to estimate the latency. We will plot, for each architecture, the fraction of objects available vs. the latency. This will be done at several scales and with several choices of critical radius.

For the competing architectures, we estimate the number of packets needed to satisfy a query as a "routing cost" as explained above.

The testbed version of this experiment goes as follows: The nine nodes are divided into two groups, of six and three. These are initially deployed so that nodes within each group are well connected, while the two groups are not connected to one another. In this configuration, the operators each use a client to publish a unique object (or maybe several objects), which are given a fixed target density of one third. Then, using the same clients, the operators query for all of the objects, and a count of those found is displayed. We certainly expect that objects will be immediately available within the group of their publisher. Now, three of the operators (arbitrarily selected) in the larger group move and join the smaller group. This should cause additional objects to become available in the (original) smaller group. This can be repeated. At each stage, the operators can use their clients to see how many objects are available. Finally, the operators scatter into small groups of three, two, or one. They each repeat the test for how many objects are available. The client also supports the experiment by telling the operators how many "good neighbors" are known, so that the nodes can be separated sufficiently to enact the scenario.

3.7.6.2 Controlling geographic density

This experiment is only meaningful for HotDiffusion – it has no analog for the competing architectures. We envision that migrating objects will leave trail markers whose time-to-live depends on the geolocation of the node. This should allow higher density of object replicas to be maintained in some regions, while a lower density will prevail in others.

We can test this idea out in simulation using the random waypoint mobility model described above. We publish a few objects, and have trail marker lifetimes depend on the distance of the node from a preferred point. We then graph the cumulative distribution of object locations over the course of the experiment. We expect that this will only work for fairly well connected networks.

To demonstrate of this effect using the wireless testbed, the nodes are arranged in a ring that is large enough so that communication can only happen along the ring, not across its diameter. The operators each choose a direction, and walk around this ring. Using their clients, they publish an object with a designated density maximum at a specific point along the ring (not necessarily near the point of publication). As they walk along the ring, the operators query for the object, which should be found preferentially near the maximum point. In post-processing of the testbed data, we graph the object density as a function of position along the ring, averaged over the course of the experiment.

4. Results and Discussion

4.1 Factors influencing availability

We considered the possibility that availability of published objects might depend on the detailed placement of the replicas. In particular, it might be beneficial to place replicas at nodes with high connectivity (high degree).

We conducted some simulations to test this idea, using random geometric graphs. We placed a fixed number of "replicas" at randomly selected nodes with different distributions constructed by assigning each node a probability proportional to a function of its degree in the graph. In each case we measured the "availability" of the object with that distribution – the fraction of nodes that could access the object as a function of the number of hops. When the function of the degree, d, is d**0 then the distribution is uniform and the samples are independent random samples. When the function is d**1, then the probability is proportional to the degree, and this is the distribution that would result from a simple random walk on the graph. We also sampled distributions generated by the weights d**2, d**-1, and d**-2. The results are shown in Figure 4.

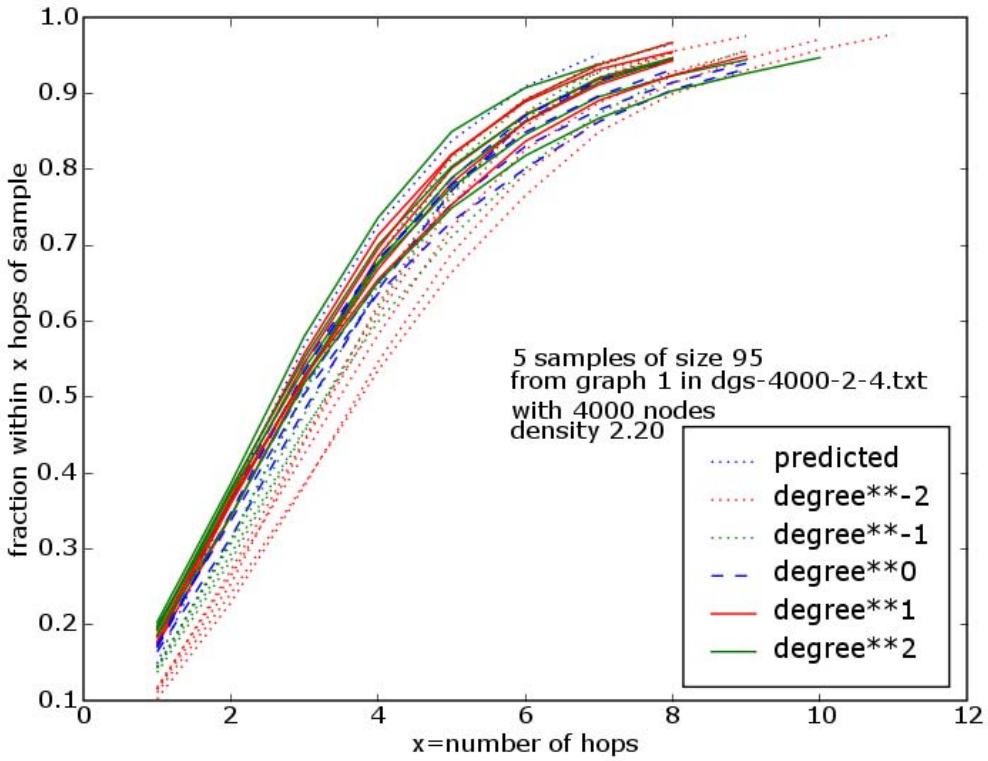

Figure 4. Availability from different distributions of object replicas[7]

We see that the samples from distributions inversely proportional to the degree provide somewhat worse availability, but the samples from distributions proportional to the degree provide roughly the same, or slightly better, availability than the independent random samples, which supports our hypothesis (that availability is not sensitive to distribution of replicas) at least for the case of random geometric graphs.

Figure 4 also contains a theoretical "predicted" curve. This curve predicts availability by computing the expected size, S(h), of the h hop neighborhood. The computation of S(h) involved computing analytically the expected size of the one hop neighborhood, S(1). If we distribute N nodes in a square of size L, then we expect

$$S(1) = N(pi/L^2 - 8/3\ L^3 + 4/30\ L^4) \quad (4)$$

This formula accounts for some edge effects in the square region. We confirmed this result experimentally (with randomly constructed examples). We also measured S(h) for

[7] In this figure, the notation "density = 2.2" is a reference to the density of nodes per unit area with the radius of connectivity normalized to 1. So, this experiment involves 4000 nodes distributed uniformly on a square of side L, such that $4000/L^2 = 2.2$. Nodes that are within one unit distance are connected. In other parts of this report we use the term "density" to refer to the number of copies of an object per node.

greater values of h. We found that to a good approximation, S(h) is proportional to $h^{1.5}$. Thus, we have:

$$S(h) = S(1)\, h^{1.5} \qquad (5)$$

Once we have S(h), we predict availability as the following probability: Distribute the K object replicas at uniformly at random among N nodes. Selecting S(h) nodes out of a total of N at random, what is the probability that at least one of those nodes will hold one of the K replicas?

$$P = 1 - [\ (N - K)/N\ ]^{S(h)} \qquad (6)$$

In this data we see a good agreement between the predicted values of availability and measured values. Thus, what seems to matter for availability is simply the size of the h hop neighborhood and the number of replicas of the object per node.

4.2 Estimating Object Density

Section 4.1 justified our belief that the central problem of HotDiffusion comes down to controlling the density of object replicas. As explained in Section 3.5, we settled on the strategy of estimating the density of object replicas in the vicinity of each object, and using that to control whether that object expired or replicated. We first looked into whether the Marker Trail algorithm (Section 3.5.1) estimates density well.

We set up a simulation of a dynamic network, using a geometric graph with uniform random waypoint model. At each waypoint, nodes were assigned random velocities uniformly in a fixed range. Using a fixed set of nodes and different connectivity radii, we were able to simulate networks with either sparse or dense connections. In this simulation, our network has 50 nodes with a static set of 10 object "replicas" and a trail length of 6. On a unit square, the connectivity radius is 0.15. The decay factor in the density estimator is 0.95. With this decay factor, about half of the estimator value depends on the last 14 steps (0.95**14 = 0.49). Figure 5 shows an example of the density estimate (of trail markers) of one of these objects as a function of time. We notice several things.

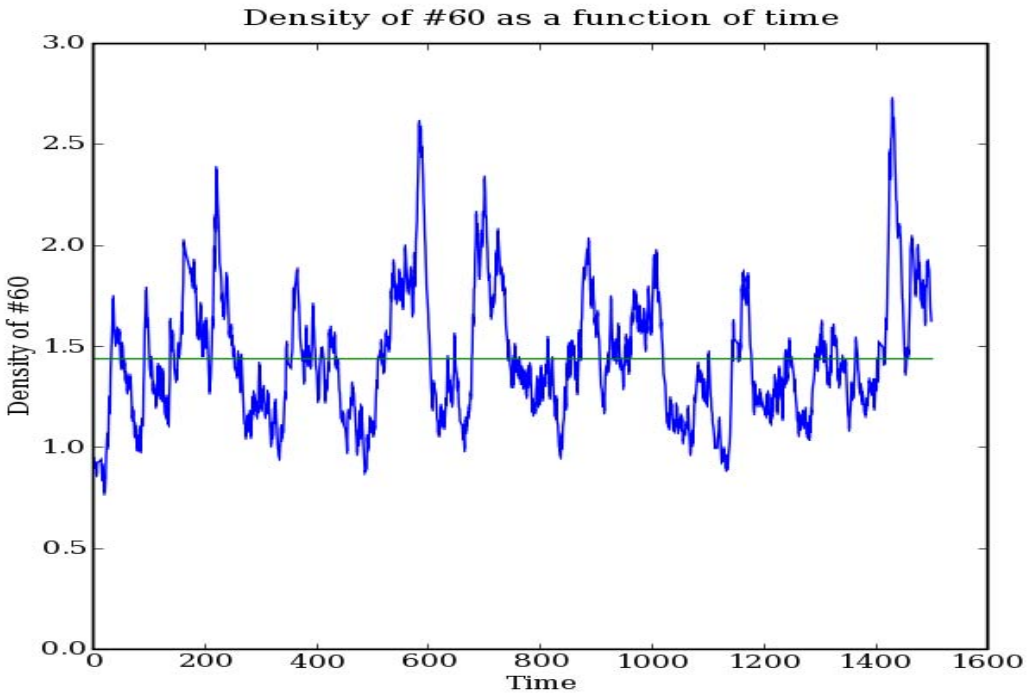

Figure 5. Typical Marker Trail density estimate as a function of time.

First, with six trail markers per object and 10 objects, the density of trail markers should have been 1.2. Instead, we see a long term average of 1.45, indicated by the horizontal line in the figure. In general we see an overestimate of the density, with the effect more pronounced in sparser networks.

To understand this effect, we add an additional object to the network, one that does not leave a trail, but does sense the trail markers of the other objects. The record of that "self-ignoring" object's density estimate is shown in Figure 6. In this figure we see that the long term average estimate is 1.2, which is the expected value. Thus, we explain the overestimate as the result of self-trail interaction, as detailed in Section 3.5.1.

The second thing we notice about both of these density estimate records is that they are quite variable. For instance, in Figure 5 the estimate ranges from about 0.9 to 2.7 while the total number of objects (and nodes) has remained constant. A detailed examination of the network configuration during times when the estimate is achieving its extreme values suggests an explanation: There are fluctuations in local density of the objects, and this estimate is correctly tracking those fluctuations. In the run represented in Figure 5 there is a sharp decline in the estimate between times 588 and 600.

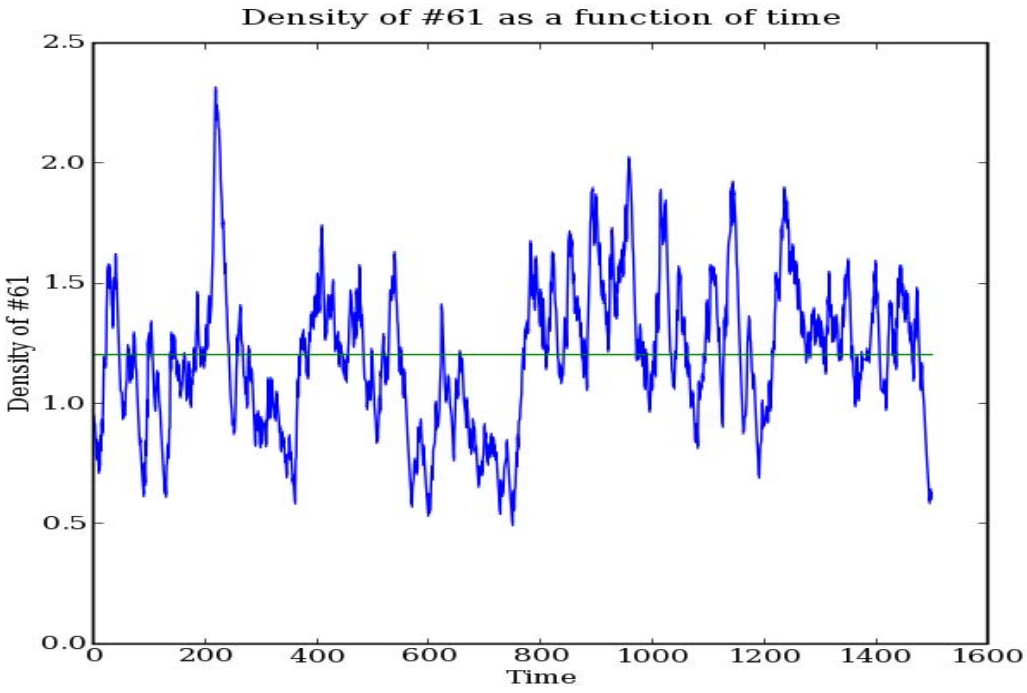

Figure 6. Density estimate of the self-ignoring object.

The network configuration at time 590 is shown in Figure 7. The object whose record we are looking at, number 60, is shown in this figure as a blue node. In this snapshot (and for some period of time before and after this) this object is isolated in a disconnected subnet that contains no other objects (shown as black nodes). Other nodes which have objects are colored black. Yellow nodes have a marker, but no object.

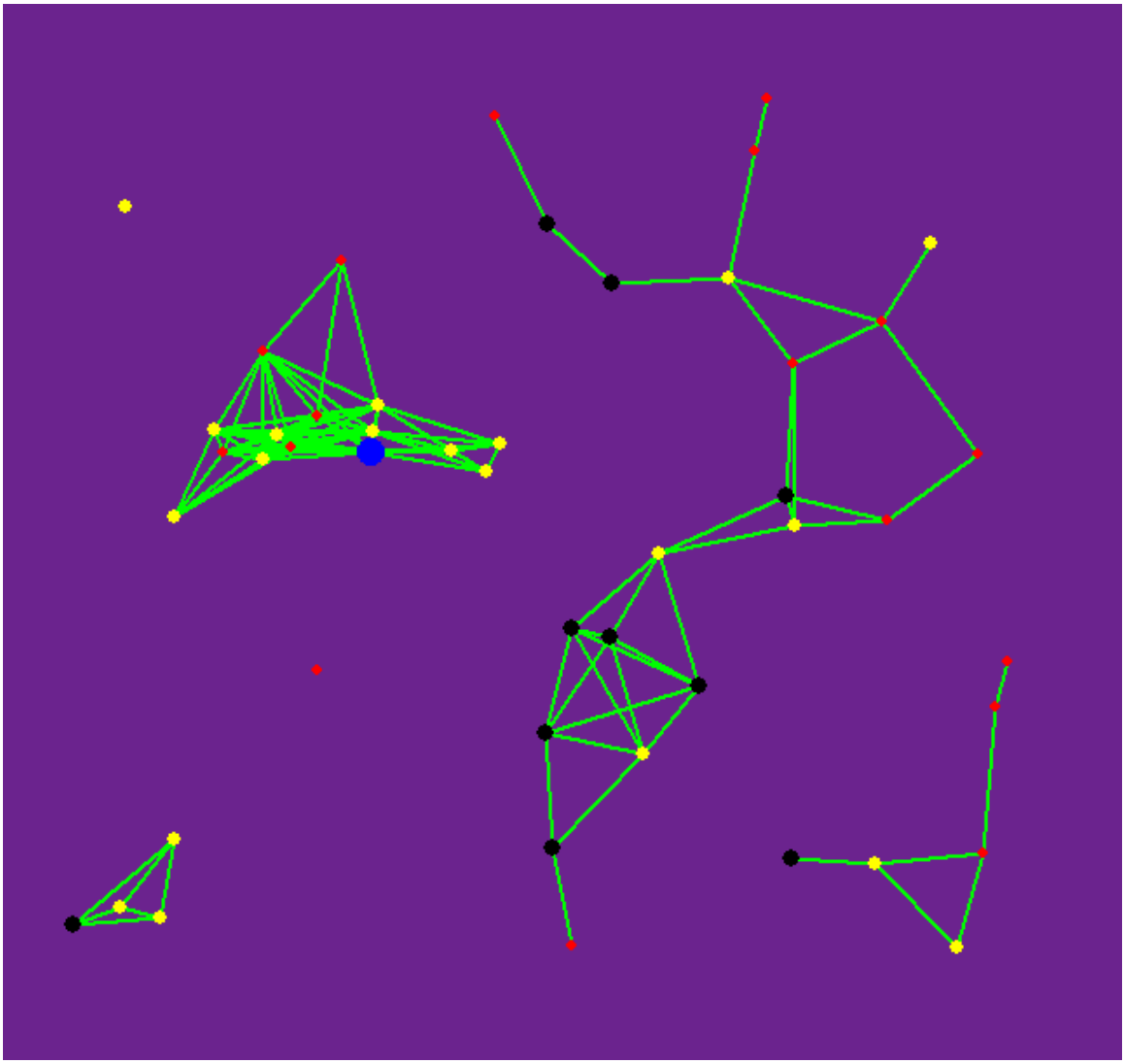

Figure 7. Network configuration at t = 590.

4.3 Feedback control of object population

We simulated using the naïve Marker Trail density estimate to control density. As described in Section 3.5.2 we arranged the trail length so that the ideal marker density would be 1.0 marker per node. If a migrating object's estimate of the density is greater than 1.0, then with some probability the object will expire at that time step. If the estimate is less than 1.0, then with some probability the object will replicate. In either case, we took the absolute difference between the estimate and 1.0, times a feedback parameter (truncated at 1.0) as the probability. A typical example showing the number of objects (MIOs) as a function of time is shown in Figure 8. The network conditions are similar to those in Figure 5.

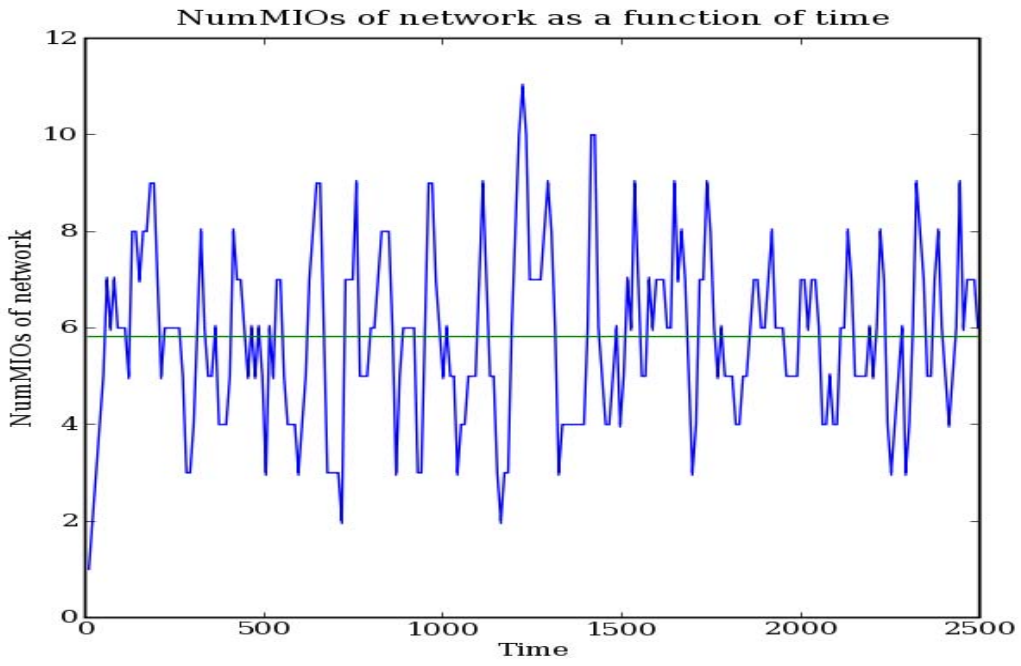

Figure 8. Object population vs. time, beginning with one object.

In this example we see the number of objects stabilizes with a long term average of 6. In this situation, with 50 nodes and a trail length of 6, we would naïvely have expected an average of 9 objects. We think that the deficit can be explained, in part, by the over estimate of the density due to self-trail interaction.

Another feature of this example is the wide fluctuations in the number of objects, ranging from 2 to 11 over the course of the run. The weaknesses of this *original* control scheme with respect to isolated subnets (which occur in this example) are explained in Section 3.5.2. The "roller coaster" effect of cloning and extinction in isolated subnets can explain some of the variability exhibited here.

4.4 Comparison of Actual and Simulated Performance

Our final HotDiffusion prototype implements the algorithms detailed in Section 3.5.2. We adjusted our simulation and analysis software to agree as closely as possible with this. This version of the simulation incorporates the following significant features:

- Asymmetric replication vs. expiration. Expiration requires that objects "collide" at a node to prevent extinction.

- Ignoring self-trail. Objects create their density estimate based only on markers from other clones, not their own markers. The length of the trail (the lifetime of markers) is adjusted so that at the target density of objects the markers will have a density of 1.0.

- Sensing neighbors, determining link quality. Nodes must remain connected for some period of time before the link can be used to transfer objects.

- One hop queries. Restricting attention to availability of objects at a maximum distance of one hop.

In these experiments we looked at two relevant statistics, comparing the simulated results to results from the implementation of HotDiffusion with an emulated wireless network. We plotted the total number of objects (called MIOs or Managed Information Objects in our figures) as a function of time, given that a fixed small number of objects are published at the beginning of the run. Ideally the population of objects should rapidly expand to "fill" the network at the desired density, and then should remain stable.

The second measurement we made was one-hop availability over time. Each of a small number of nodes publishes a unique object. Then, at regular intervals each of those nodes queries for all of the objects – to see what fraction can actually be retrieved. If an object is not available at some moment, then we can envision querying repeatedly until it is found, since the motion of nodes and the migration of objects may bring a copy near. The relevant quantity is the time delay – the earliest time at which the object is available. We then plot the availability (fraction of objects that can be reached) vs. the time delay, averaging measurements over the whole run.

We used the node-to-node connectivity log from the simulation to drive the emulator so that network conditions were identical. In these networks (simulated and thus emulated) each link is either present in both directions (link quality 1.0) or absent (link quality 0.0) – there is no link asymmetry and no "poor quality" links. In spite of this, we continued to use (and simulate) the link quality estimation mechanism. The need to do link discovery and link quality estimation has a major impact on performance, as described below.

Figure 9 shows the time evolution of a population of objects, in corresponding simulated and real networks, over a one hour time span, beginning with 10 distinct objects. In this case the network had 20 nodes with a radius of connectivity of 0.15. The target density is 0.33 objects per node. The settings of our link quality estimator (beacon interval 6 seconds; window size 9; threshold 0.85) mean that HotDiffusion will take 48 seconds to recognize that a link is "good" in both directions. The nodes are moving with randomly selected speeds in the range of 0.005 to 0.0005 units per second, where the square that contains all of the nodes has size one unit. The time interval between object moves is 7 seconds.

Since there are ten distinct objects, each seeking a density of 0.33 on a network of 20 nodes, we expect the total number of objects in the system to stabilize at around 66. The simulated and real traces do appear to rise together and level off after about 2500 seconds at very roughly that level. A good deal of noise is evident in the totals. There is some difference between the two traces which may not be significant.

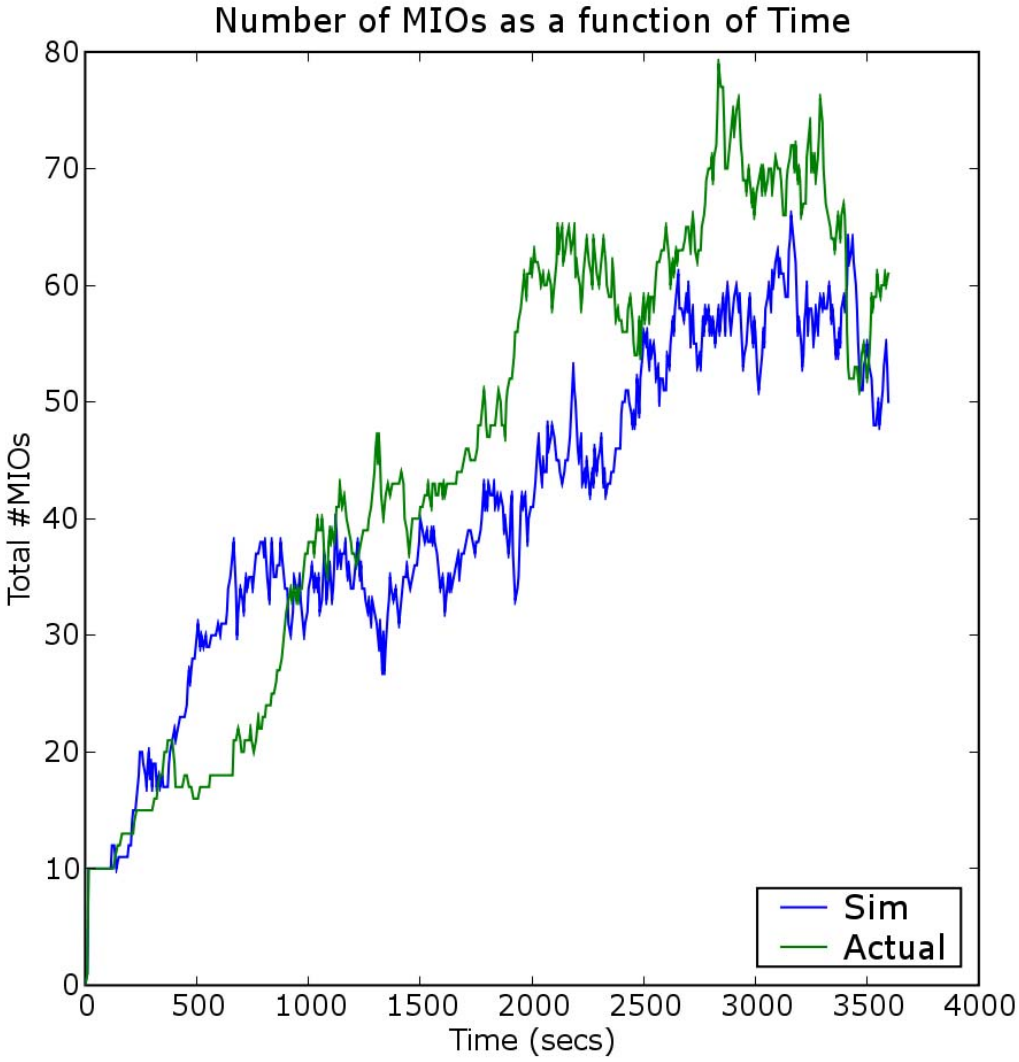

Figure 9. Number of objects over a one hour run, simulated compared with real.

One feature of this data which stands out is the length of time necessary for the object population to stabilize. We attribute this to the sparseness of this network. With a connection radius of 0.15 the network typically ranges between 15 and 25 links over time. Contrast this with the 380 bi-directional links that are possible in a 20 node network. Given the distance scale of this network and typical node speeds, it follows that nodes become connected for time periods (typically) ranging from 10s to 100s of seconds. HotDiffusion only uses links that are recognized as "good" – recognition that takes roughly 48 seconds. Thus, with these parameters, HotDiffusion will only be using a fraction of the available connections to propagate objects.

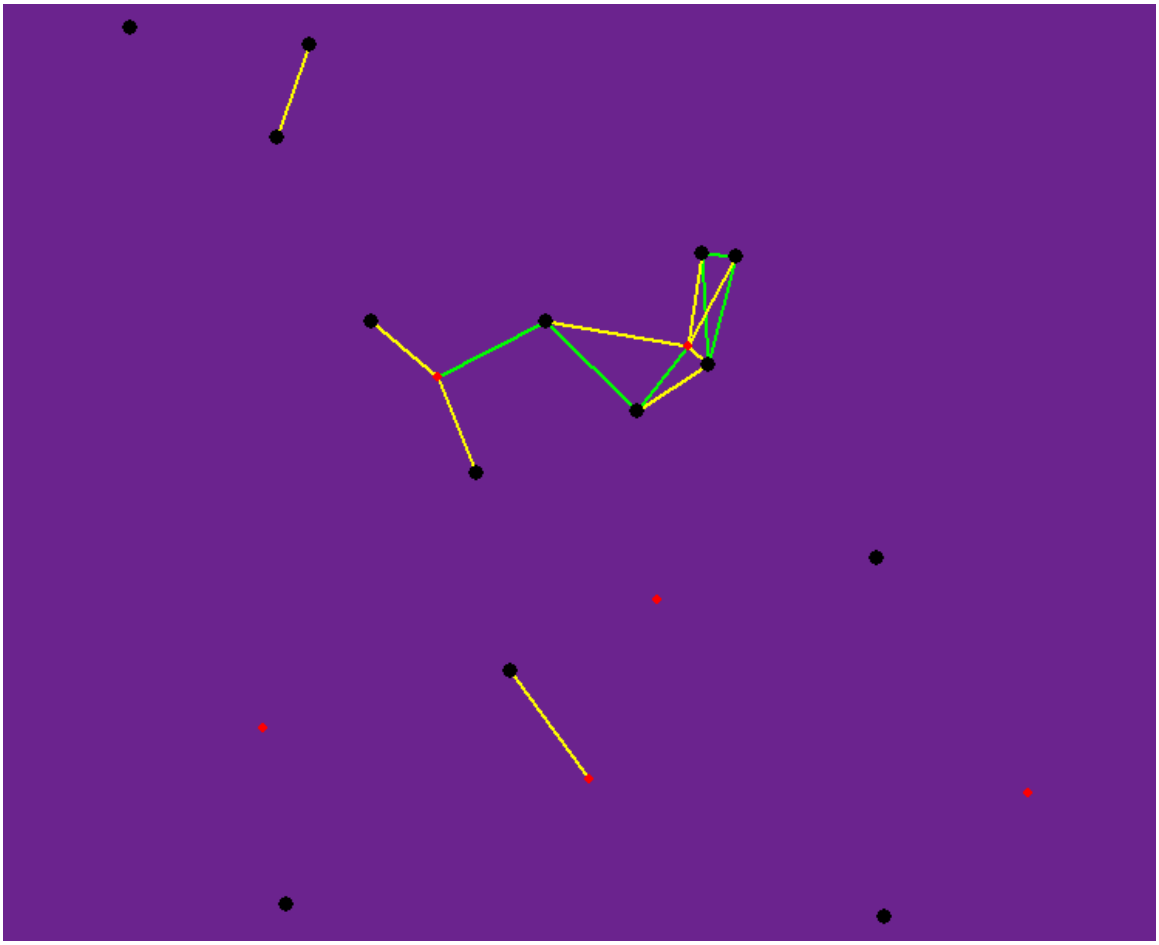

Figure 10. Snapshot of network showing connections in yellow and "good" connections in green.

Figure 10 shows the arrangement of nodes and links during this experimental run at an arbitrarily selected moment (about 612 seconds into the run). The arcs drawn in green are the ones that HotDiffusion recognizes as good – suitable for object migration. In this example, which is typical, many objects (residing at nodes shown in black) are at nodes that are effectively isolated, since they have no good links (i.e., no green arcs).

To see the impact of this link quality measurement delay, we can re-run the simulation with the delay set to about 2 seconds, so that almost all links are used by HotDiffusion. The results are shown in Figure 11. The object population levels off after only 500 seconds. The steady-state level is about 40 objects, less than our expected value of 66. The reason for this discrepancy is not currently understood. This is still a sparse network with many isolated nodes. If the connection radius is further increased to 0.45 (three times as great) with the network then forming a single connected component, the stable value is higher – about 50 – and this number is first achieved after only 250 seconds.

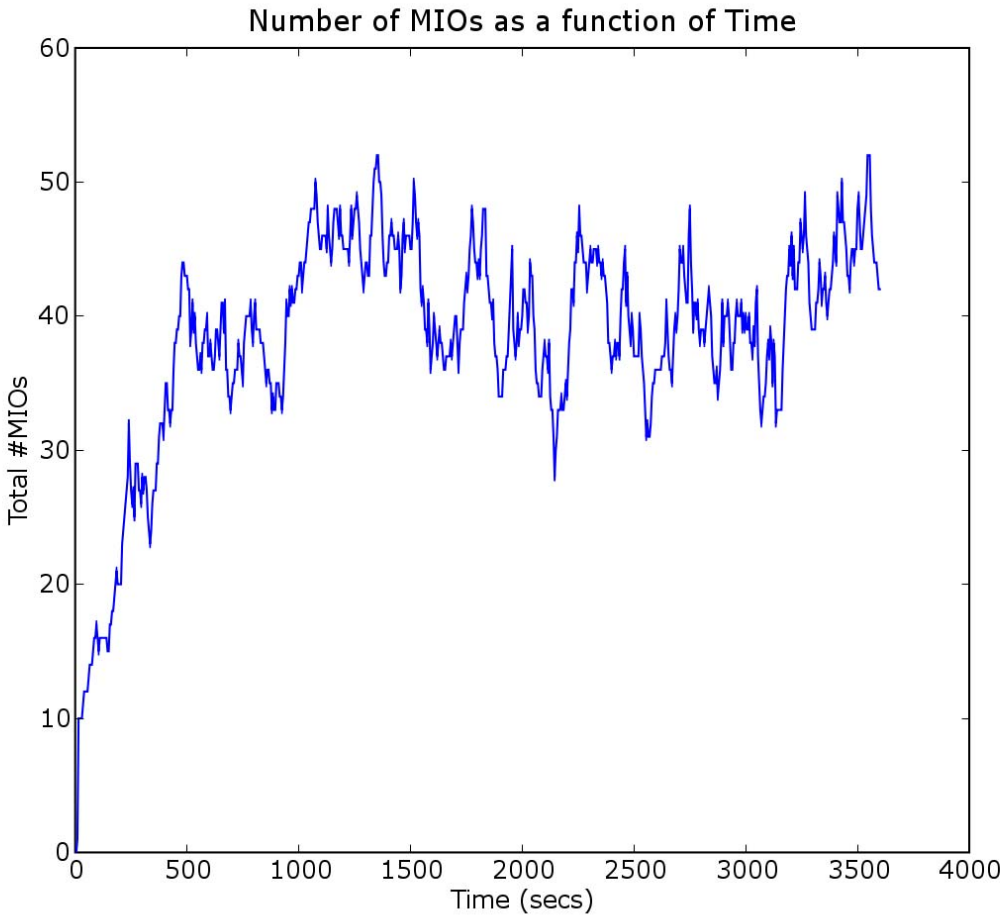

Figure 11. Simulated object population with very rapid link quality estimation.

In this same pair of experimental runs, we compare availability. Ten of the nodes each publish unique objects. We then wait until the population of objects has first reached its average value, then measure availability versus latency, as described in Section 3.7.6.1. In Figure 12, the dark blue curve shows simulated HotDiffusion availability, while the light blue curve shows availability measured with the prototype. As indicated in Figure 9, in this run objects are diffusing quite slowly so that the average population occurs at a time when numbers of objects are still growing. For querying, HotDiffusion uses broadcasts, which are not restricted to "good" links. For both simulated and real experimental runs, objects migrate across "good" links, but availability of those objects is measured using the underlying true connectivity, which is much less sparse. The

simulated and real curves show some difference.

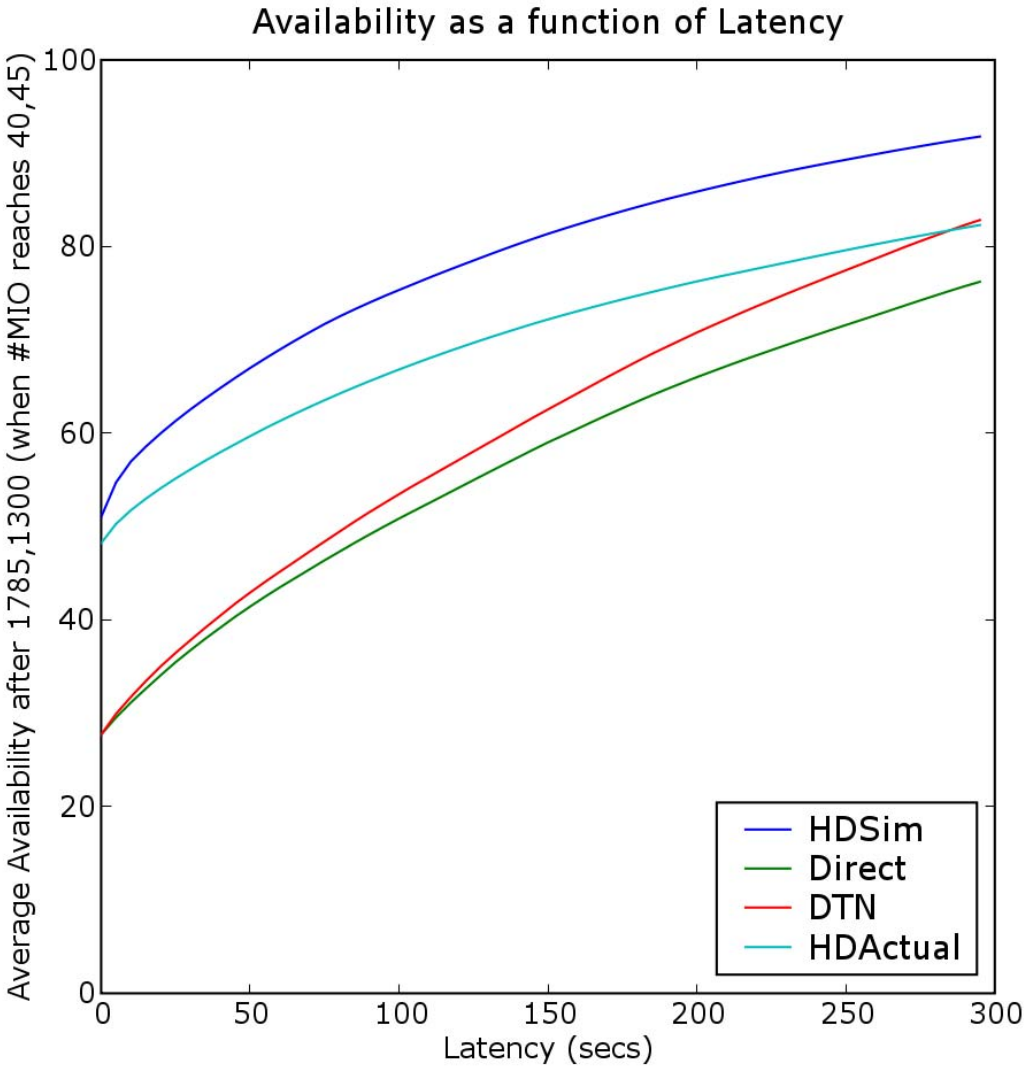

Figure 12. Percent Availability vs. Latency for HotDiffusion (simulated and real) , Direct, and DTN.

This same plot contains curves for two competing information management architectures. These are explained in Section 3.7.3. The green curve ("Direct") shows availability based on the client being able to form a complete ad hoc connection to the server node (in this case, that is the publisher node). The red curve ("DTN") is an idealization of using Disruption Tolerant Networking (DTN) to mediate the client-server interaction.

Both of these idealized methods are given the advantage of working perfectly – always finding the best route, if it exists, without cost. In the case of DTN, that assumes some degree of prescience! These networks use the underlying true connectivity without delay,

and thus see a much less sparse network. They also have one other key advantage: the ten publisher nodes are the same as the ten query nodes. Since objects remain at the publisher node, they will always be available there! Thus, in this test, those methods are, in effect, given 10% availability for free. In contrast, objects in HotDiffusion move every few seconds. Thus, a short time after an object is published, the publisher node will not have any advantage in querying for it.

This figure *shows a considerable advantage* for HotDiffusion, simulated or real, over both of the competing architectures, in spite of the 10% head start and other advantages that they are given in their simulations.

We can also compare HotDiffusion and the other architectures in the presence of a more conspicuous network impediment. In the following experimental runs we again used 20 nodes, but with a connection radius of 0.24. However, in these runs we divided the area into four quadrants. While nodes are allowed to follow paths that cross from one quadrant to another, links are only formed within each quadrant. To illustrate this, Figure 13 is a typical snapshot of this network, with the quadrant divisions marked.

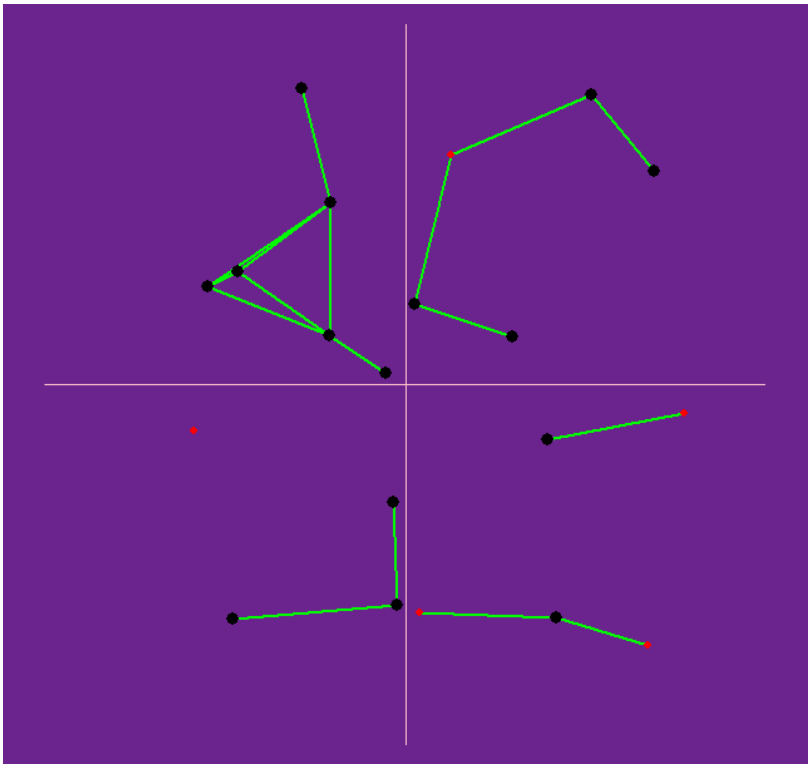

Figure 13. Network snapshot for disconnected quadrants.

The only way for communication to happen between quadrants is for nodes to move from one to the other. This restriction impacts the three architectures differently. For the direct access scheme, nodes must simply wait until the desired object is in the same quadrant as they are. DTN uses the motion of nodes to carry requests across the quadrant boundaries, and to carry the replies back. In contrast, HotDiffusion uses node motion to

carry information object replicas into all of the quadrants where, ideally, a stable population will be established. The immediate response to a HotDiffusion query only depends on objects that have already migrated into the vicinity of the query node. Over time, if the query is repeated, some additional objects may become available due to further migrations and node motion. In this experiment we again published 10 objects initially. The availability results from this setup are shown in Figure 14.

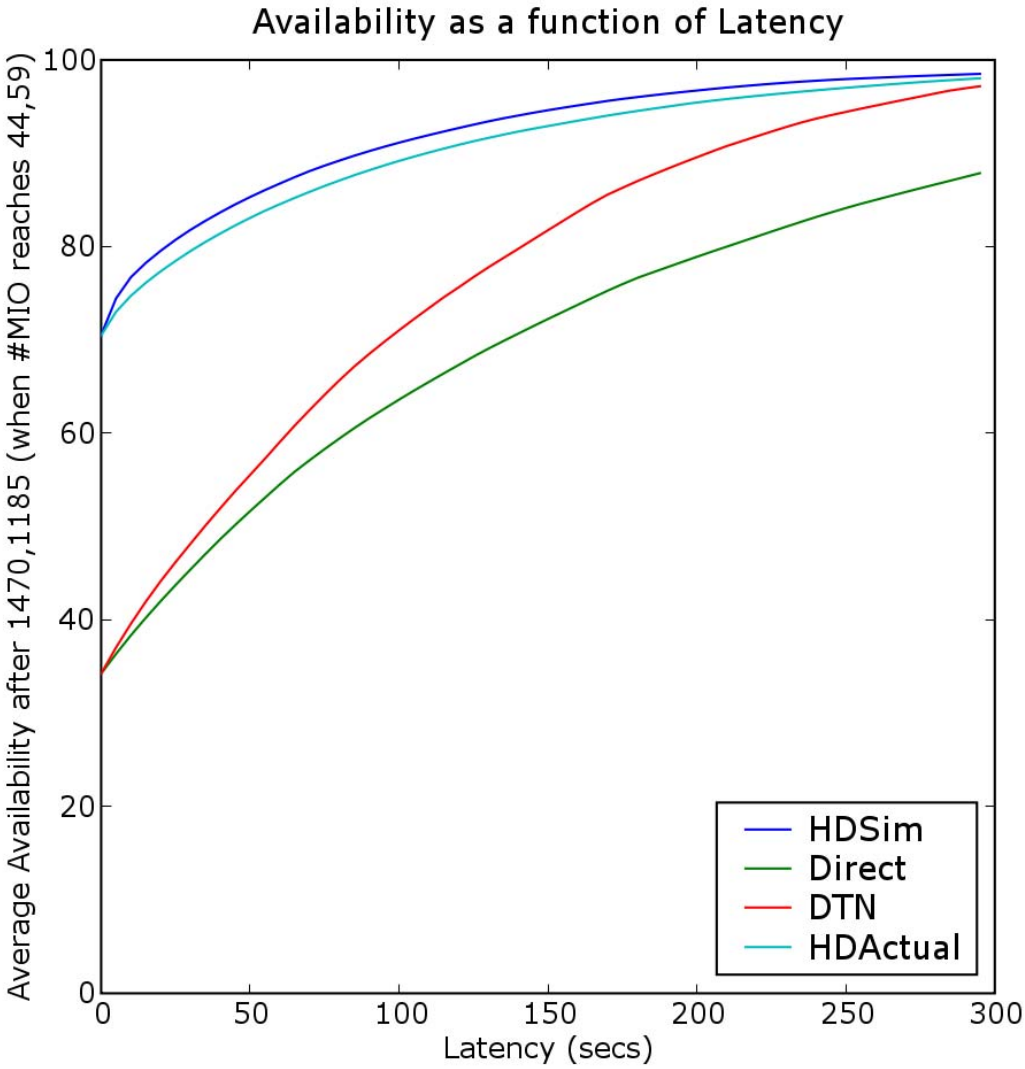

Figure 14. Availability vs. Latency for Disconnected Quadrants

This figure shows good agreement between the simulated and real runs of HotDiffusion. The advantage of HotDiffusion over the competing architectures, especially at small

latency, is dramatic. As in the previous experiment, both Direct and DTN are given 10% of availability "for free" in their simulations.

4.5 Experiments at a larger scale

We ran an experiment with 100 nodes. In this run we used parameters intended for a scaled-up version of the network used in Figures 9, 10, and 12. Since there are 5 times as many nodes, we reduced the connection radius by a factor of the square root of 5 (roughly, 2.24). We also slowed down the nodes by this same factor, in order to make the make the interactions between connected nodes have the same temporal distribution.

This kind of simple geometric scaling should approximately reproduce local network properties in networks of different size. However, there are large-scale and global effects that do not scale in this way. In particular, in the limit of large network size, N, the *critical radius* is asymptotically $r = 4 \log(N)/N$. The critical radius is that connection radius for which the random geometric graph network will have a single connected component with probability one.

With 10 objects published initially, our ideal steady-state population is 333 objects. At this scale we only simulate HotDiffusion and the other architectures. Figure 15 shows the growth of the population of objects over time. In this instance, the number of objects appears to grow fairly steadily over the course of one hour, and does not reach the target density during the experiment. We again attribute this slow dispersion to the extremely sparse connections in this network, especially when the effect of delays for link quality estimation is considered, as with the earlier example on which this one is modeled.

In this same experimental run we measured the average availability versus latency for HotDiffusion and the two competing architectures. The results are shown in Figure 16. In this run, HotDiffusion provides much greater availability than competing architectures, especially at low latency. Overall availability using HotDiffusion is lower in this run than in the smaller example on which it is modeled, because the population of objects is significantly lower than the target value during the whole run. As in the previous example, the two competing architectures are given 10% of availability "for free" and can make use of the full connectivity of the network.

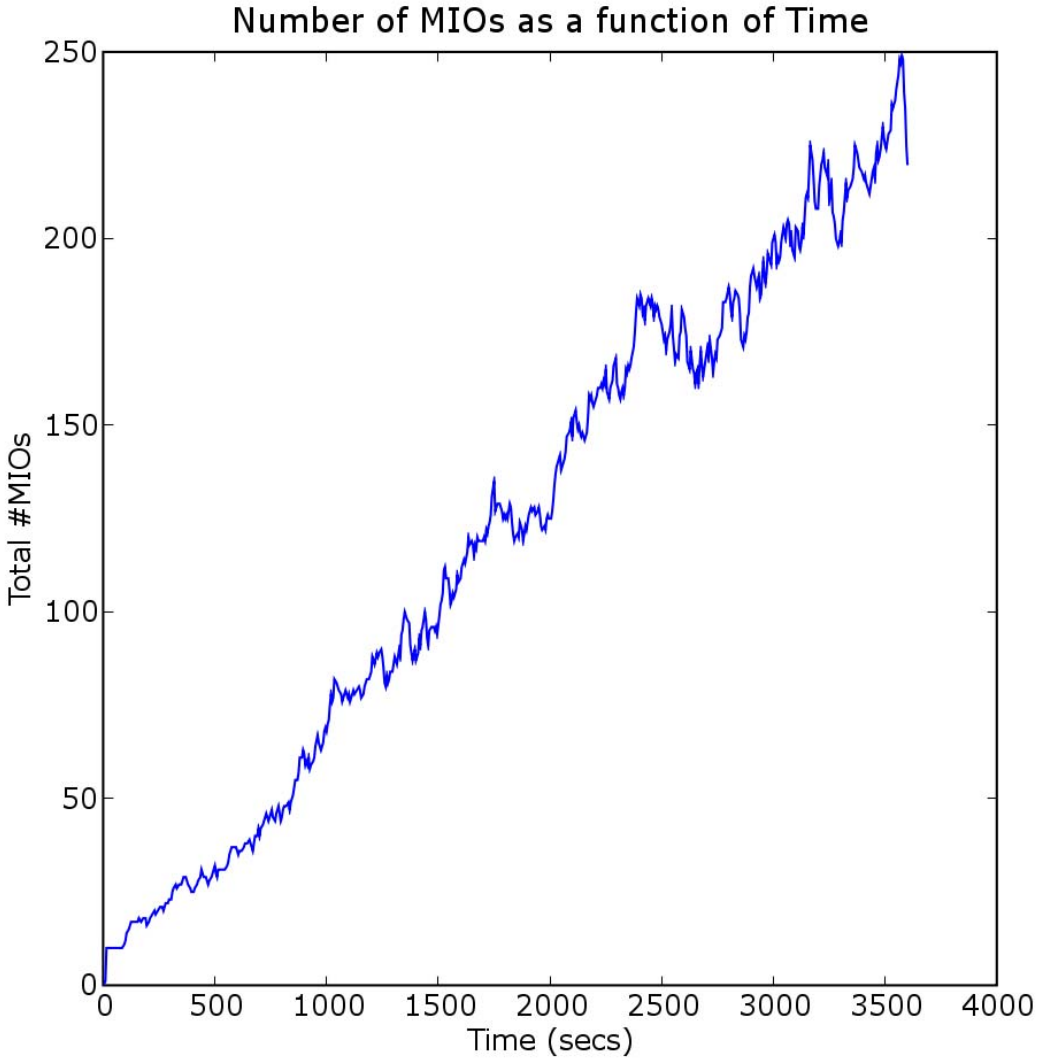

Figure 15. Population of objects over time in a 100 node network.

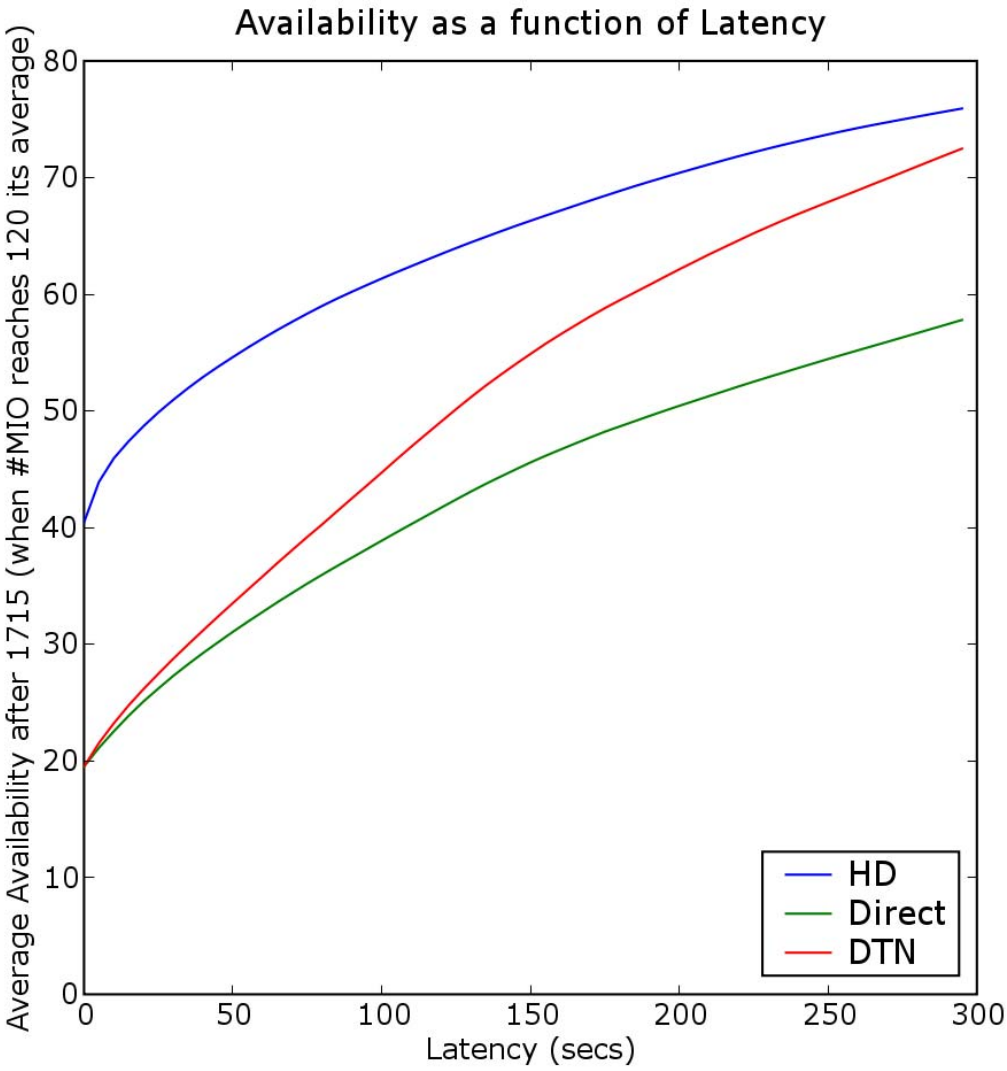

Figure 16. Availability vs. latency in a 100 node network.

5. Conclusions

Ad hoc tactical networks are characterized by poor connectivity, highly dynamic and unpredictable topology, and low-capability nodes. Conventional distributed information systems cannot function effectively under these circumstances. Yet, tactical users (and un-manned nodes) will clearly need something resembling an information system in the near future in order to fulfill their projected roles. This gap motivates the HotDiffusion project.

The goals of the HotDiffusion effort were to validate and develop a novel peer-to-peer concept for providing information management services in *ad hoc* tactical networks. At its core HotDiffusion relies on an analogy with thermal physics to disperse information objects among the peer nodes by a process analogous to diffusion. More specifically, the project aimed to determine the *feasibility* of HotDiffusion, to uncover the necessary *engineering principles*, and to design, build and experiment with a *prototype*. The major hurdles or questions we faced in this effort were:

1. Given that copies of published information objects will be distributed among the nodes via diffusion, and will be retrieved only from nearby nodes, what distributions of objects are desirable?

2. How can the diffusion, replication, and expiration of information objects be controlled locally so that the desired distribution emerges as a steady-state?

3. How well will an information system based on this concept perform under a range of network conditions? What are the strengths of HotDiffusion?

4. What additional engineering challenges must be met to implement a working prototype information system based on these ideas?

We now have results that bear on each of these questions. As a consequence, we know that HotDiffusion is feasible and can be advantageous in certain challenging circumstances. We also have a working implementation of HotDiffusion which runs on a testbed of handheld wireless nodes, a system for emulating larger HotDiffusion networks, and a corresponding simulation. These form a suitable basis for further experimentation and demonstrations.

In answer to the **first question**, we found that availability could be predicted from a simple model involving the density of replicas and the number of nodes that are expected to receive the query (i.e., the size of the neighborhood out to h hops). Thus, to ensure availability, it suffices to control the density of replicas in relationship to the (typical) neighborhood size. As shown in Section 4.1, other factors, in particular, the degree of connectivity of the nodes holding replicas, are largely irrelevant, except in so far as they affect neighborhood size.

Besides availability, other desirable properties such as latency, integrity, and graceful degradation of service, are all expressible in terms of availability, and thus all dependent

on density in some form. Costs in HotDiffusion will go up at least linearly in density, so that controlling density – as opposed to simply ensuring sufficient density – is essential.

With regard to the **second question**, we showed that the Marker Trail algorithm could be used to make local decisions about replication and expiration of objects. The "trail length" or time-to-live of trail markers defines the target density of objects. This method, in effect, sets up a feedback loop to control object density. We found that several enhancements to our original straightforward scheme were necessary for good performance in sparse networks.

We combined the Marker Trail control scheme with an "unbiased" random walk. In this combination, we found that under some conditions, the object population quickly grew to the target level and remained roughly steady (e.g., Figure11 and Figure 8)[8]. Under other conditions, we found that the population of objects grew slowly, and suffered from large fluctuations (e.g., Figure 9 and Figure 15). Generally, these were sparser networks with strong barriers to the spread of information by any strategy.

To address the **third question**, we measured HotDiffusion performance (as availability vs. latency) in several test runs with an emulated network, as well as corresponding simulated runs (e.g., Figure 12 and Figure 14). In these same networks[9] we simulated an idealized version of two competing information management architectures and measured the same statistic. HotDiffusion with emulated network and its corresponding simulation exhibit similar but not identical performance. In these runs, representing different challenging circumstances, HotDiffusion outperforms the two competing architectures. The run depicted in Figure 12 has a very sparse network. In Figure 14, the network is much better connected, but is artificially segmented into quadrants. In this latter example, the only communication between quadrants occurs when a node "ferries" an object (or a DTN message) across the boundary.

In some other comparison runs, where the network is relatively well connected, HotDiffusion does not do as well as the competing architectures. With most nodes in a single connected component most of the time, both alternatives should provide nearly perfect availability immediately. In this setting, HotDiffusion's probabilistic approach works against it, reducing performance somewhat. Yet, it is encouraging that HotDiffusion continues to give service under conditions outside its optimal range.

This comparison is not really a level playing field, due to several factors. We summarize the relative advantages and disadvantages that each architecture had in these tests:

- HotDiffusion had the disadvantage that it could only use bi-directional "good" connections for object migration. Since measuring link quality takes time for the

[8] Figure 8 shows data from a run that used our earlier un-enhanced feedback method.

[9] Each quartet of comparable runs, one emulated and three simulated, used the identical underlying node connectivity, which was generated once, and replayed.

exchange of beacons, HotDiffusion was effectively restricted to a much sparser network (see Figure 10).

- HotDiffusion had the advantage that data is replicated, while the competing architectures are not. In retrospect, replication seems important when networks are fragmented.

- HotDiffusion was given the advantage that querying did not begin until some time after publication – to give objects a chance to diffuse and replicate. In contrast, this delay was of no benefit (or harm) to the competing architectures.

- Both of the competing architectures were given a (roughly) 10% boost in availability because the "query nodes" were the same as the publishers.

- Both of the competing architectures were assumed to "route" messages perfectly and instantly within any connected component of the network.

- DTN was assumed to have "prescience" – to be able to predict the *optimal* node to advance a message to at each stage, so as to reach the destination at the earliest moment.

With respect to the **fourth question**, implementing the HotDiffusion prototype, and experimenting with that implementation, revealed several important considerations:

In our early simulation, we simply assumed that each pair of nodes was either connected, or not, and that the nodes were aware of these links. In parallel we developed software (the *neighbors'* module) to detect neighbors, measure link quality in both directions, and apply a threshold to those measurements. With this software, there is necessarily a delay between the changes in link status, and the recognition of that change in the system. We did not recognize how important it is to minimize this delay – to make it much shorter than typical connection lifetimes. Significant delays can cause the system to, in effect, use a much sparser and more error prone network than they really have. This occurs because the nodes avoid some new links that are good, while continuing to use links that are no longer good.

It turns out that by tuning the parameters in our neighbors' module we can speed up the recognition of link changes. However, we did not recognize the need to do this until late in the project, by which time, this was no longer practical. Such changes will entail some costs associated with more rapid BEACON messages – congestion and additional event processing. We can also be somewhat less accurate in our measurement of link quality, thus requiring less data.

When implementing HotDiffusion we needed to decide between fixed size data structures and dynamically allocated ones on several occasions. For example, should the object cache be fixed or dynamic? If it is fixed size, then what should be done if it fills up? If it is dynamic, then should it grow without bound? There is not really a natural answer in

either case. For the object cache, we made it fixed, and much larger than we anticipated needing for our experiments. However, that is not a realistic solution.

Any deployed version of HotDiffusion will need to address this tension between the dynamism of the network and information space, and the finiteness of memory resources. This may be partially handled at the application level by implementing objects with finite lifetimes, as we did in our system. However, this is not a complete solution.

In our prototype, we implemented the node-to-node links with UDP over ad hoc mode 802.11g. In addition to allowing for easy, user-space programming, this service had several useful capabilities. Any implementation of HotDiffusion over more primitive packet radio links would probably need to consider implementing something like these capabilities:

- Checksumming for detection of bit-level errors
- Delivery of packets to specified peers that are in range, as well as broadcast of packets to all nodes within range.
- Delivery of packets to specified ports (or to specified protocols). This allows HotDiffusion to coexist with other network services.
- Fragmentation of large messages, and reassembly.

We must acknowledge that several puzzles remain in our data. Most notably: We confirmed that a "self-ignoring" Marker Trail gives the expected answer for density. Yet, when this self-ignoring Marker Trail signal is used in a feedback loop to control density, the density appears to stabilize at the wrong equilibrium value – always a bit too low.

There are some differences between the simulated and emulated performance measures for HotDiffusion, in experimental runs that should be exactly comparable. We do not have enough data to know if these differences are significant or not.

To summarize, HotDiffusion appears to excel under very sparse network conditions, especially if there is enough dynamism in connectivity. We expect its performance to degrade gracefully as conditions worsen. It also functions, though not optimally, under good network conditions. HotDiffusion will opportunistically "ferry" information on nodes moving between disconnected sub-networks. It does this without planning or central coordination.

On theoretical grounds, we expect HotDiffusion to excel in circumstances where node motion and connectivity are less predictable, and where the quality of the network varies greatly over time, and geographically.

6. Recommendations

HotDiffusion stands at one extreme of a spectrum of techniques for information services in tactical networks. Under some network conditions it can deliver significant benefits over other methods. We can advance the overall aim of information management by further testing and refinement of HotDiffusion, particularly in these areas:

- Test HotDiffusion under more realistic network models (e.g., networks with correlated node motion, and where nodes are heterogeneous in their behavior). Design enhancements to HotDiffusion to address these conditions, as needed.

- Strengthen and enhance our HotDiffusion implementation. Fix several known shortcomings such as the fixed object cache size.

- Consider implementing a *subscribe* operation.

In HotDiffusion, queries are answered probabilistically, and the dispersion of information may take a considerable amount of time. Designing applications to make good use of this kind of service will require some innovation. For example, an application which depends on receiving "the most recent" object of a certain kind, will not be able to naively query and then select the most recent returned value. The application can only reasonably decide retrospectively, that the object was likely the most recent when it was received. As another example, node memory constraints strongly suggest the need for application-level strategies for pruning the information space. Therefore we recommend development of techniques for effective programming with this new service, many of which would benefit any tactical information system. Our current HotDiffusion implementation, running on an emulated network, would be a good basis for experimenting with these techniques.

With our current system, applications (and administrators) may "shape" the information space by publishing new objects, and by specifying a finite lifetime for objects *when they are published*. Yet other forms of control of the information space, such as deletion of objects, seem desirable. Implementing these will not be straightforward, given that once an object is published, it would be difficult to find all of its replicas.

We believe that a reasonable approximation to deletion can be implemented via *Destructor* objects. A Destructor object (or more generally, a Mutator object) is a published information object that diffuses just like any other. Each Destructor object is intended to delete one or more target objects, identified by a predicate that the Destructor carries in its metadata. When a Destructor meets other objects at a node, the predicate is evaluated, and if the other object is a match, it is deleted. *Mutator* objects, are like Destructors, but can take more general actions on matching objects.

Mutator and Destructor objects are a very natural evolution in functionality for HotDiffusion. Our current implementation and simulation can readily be enhanced to include this feature. This would provide a basis for experimentation with both the mechanics of such objects, and with the best way to program applications using them.

One aspect of HotDiffusion that proved to be somewhat brittle is the notion of a "good neighbor." Objects migrate across node-to-node links that have good quality in both directions. This is because we want to be reasonably certain that the object will arrive at its destination, and that an acknowledgement will be delivered to the sender. Without this, objects might get lost in transit, or become spuriously replicated by half-finished migration steps. Additionally, determining who the good neighbors of a node are proved to be expensive and introduced a harmful delay in using new links.

Can we build a system that resembles HotDiffusion but avoids these problems? We envision *PlasmaDiffusion*, a system which will only use broadcast transmissions – no point-to-point links. As we conceive it, nodes will not know the identity of their nearby peers, and will never address a message to a specific peer. Even querying within the local neighborhood of nodes will use only broadcasts. The main difference between PlasmaDiffusion and our current system is that in HotDiffusion individual object replicas migrate – they are conserved except when replicating or expiring, explicit events that we control. In contrast PlasmaDiffusion has an underlying model that is closer to epidemiology – the spreading of objects is a sort of contagion.

PlasmaDiffusion would be the next generation of HotDiffusion, combining the benefits of HotDiffusion with an underlying network layer designed to handle real-world problems encountered in wireless networks. PlasmaDiffusion would offer the flexibility of highly available message information objects in a dynamic mobile ad hoc network using totally distributed self organization for object migration and replication. It would be a more resilient architecture than HotDiffusion, making fewer assumptions about the underlying wireless links, and providing an ultimate fallback algorithm for distributed information platforms. This would be well-suited for manned and unmanned nodes in a deployed tactical network in which information objects or reports are periodically published.

7. References

Adelstein, Frank, Sandeep KS Gupta, Golden Richard III, and Loren Schwiebert, "Fundamentals of Mobile and Pervasive Computing," McGraw-Hill Professional, 2004.

Anderson, Don. "Gumstix, inc. white paper 'a one year report'", Palo Alto, CA, May 31, 2005. http://gumstix.com/press/gumstix_One_Year.pdf

Bonney, Jordan, Glenn Bowering, Ryan Marotz, and Kirk Swanson, "Agent-Based Space Network Emulator (ABSNE)", 2008 IEEE Aerospace Conference," March 1-8, 2008.

Farrell, Stephen and Vinny Cahill. Delay- and Disruption-Tolerant Networking. Artech House Publishers, 2006.

http://www.infospherics.org/

"Report on Building the Joint Battlespace Infosphere", USAF Scientific Advisory Board Report, December 17, 1999.